walter DOBERENZ
thomas GEWINNUS

VISUAL C# 2012

GRUNDLAGEN UND PROFIWISSEN

// C#-Grundlagen, Techniken, OOP
// GUI-Programmierung mit Windows Forms und WPF
// Entwickeln von Windows Store Apps

E-Book

EXTRA: Kostenloses E-Book inkl. 1000 Seiten Bonuskapitel

HANSER

Doberenz/Gewinnus

Visual C# 2012
Grundlagen und Profiwissen

Bleiben Sie auf dem Laufenden!

Der Hanser Computerbuch-Newsletter informiert Sie regelmäßig über neue Bücher und Termine aus den verschiedenen Bereichen der IT. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter

www.hanser-fachbuch.de/newsletter

Walter Doberenz
Thomas Gewinnus

Visual C# 2012

Grundlagen und Profiwissen

HANSER

Die Autoren:

Professor Dr.-Ing. habil. Walter Doberenz, Wintersdorf

Dipl.-Ing. Thomas Gewinnus, Frankfurt/Oder

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <*http://dnb.d-nb.de*> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdrucks und der Vervielfältigung des Buches, oder Teilen daraus, sind vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2013 Carl Hanser Verlag München
http://www.hanser-fachbuch.de

Lektorat: Sieglinde Schärl
Herstellung: Thomas Gerhardy
Satz: Ingenieurbüro Gewinnus
Sprachlektorat: Walter Doberenz
Umschlagdesign: Marc Müller-Bremer, *www.rebranding.de,* München
Umschlagrealisation: Stephan Rönigk
Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

ISBN 978-3-446-43439-4

E-Book-ISBN 978-3-446-43521-6

Inhaltsverzeichnis

Vorwort **51**

Teil I: Grundlagen

1 Einstieg in Visual Studio 2012 ..... **57**

1.1 Die Installation von Visual Studio 2012 57
1.1.1 Überblick über die Produktpalette 57
1.1.2 Anforderungen an Hard- und Software 59
1.2 Unser allererstes C#-Programm 60
1.2.1 Vorbereitungen 60
1.2.2 Quellcode schreiben 62
1.2.3 Programm kompilieren und testen 63
1.2.4 Einige Erläuterungen zum Quellcode 63
1.2.5 Konsolenanwendungen sind langweilig 64
1.3 Die Windows-Philosophie 65
1.3.1 Mensch-Rechner-Dialog 65
1.3.2 Objekt- und ereignisorientierte Programmierung 65
1.3.3 Programmieren mit Visual Studio 2012 67
1.4 Die Entwicklungsumgebung Visual Studio 2012 68
1.4.1 Neues Projekt 68
1.4.2 Die wichtigsten Fenster 69
1.5 Microsofts .NET-Technologie 72
1.5.1 Zur Geschichte von .NET 72
1.5.2 .NET-Features und Begriffe 75
1.6 Wichtige Neuigkeiten in Visual Studio 2012 82
1.6.1 Die neue Visual Studio 2012 Entwicklungsumgebung 82
1.6.2 Neuheiten im .NET Framework 4.5 85
1.6.3 C# 5.0 – Sprache und Compiler 87

1.7 Praxisbeispiele ... 87
1.7.1 Windows Forms-Anwendung für Einsteiger ... 87
1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger ... 92

2 Grundlagen der Sprache C# ... 101

2.1 Grundbegriffe ... 101
2.1.1 Anweisungen ... 101
2.1.2 Bezeichner ... 102
2.1.3 Schlüsselwörter ... 103
2.1.4 Kommentare ... 103
2.2 Datentypen, Variablen und Konstanten ... 104
2.2.1 Fundamentale Typen ... 104
2.2.2 Wertetypen versus Verweistypen ... 105
2.2.3 Benennung von Variablen ... 106
2.2.4 Deklaration von Variablen ... 106
2.2.5 Typsuffixe ... 108
2.2.6 Zeichen und Zeichenketten ... 109
2.2.7 object-Datentyp ... 111
2.2.8 Konstanten deklarieren ... 111
2.2.9 Nullable Types ... 112
2.2.10 Typinferenz ... 113
2.2.11 Gültigkeitsbereiche und Sichtbarkeit ... 114
2.3 Konvertieren von Datentypen ... 114
2.3.1 Implizite und explizite Konvertierung ... 114
2.3.2 Welcher Datentyp passt zu welchem? ... 116
2.3.3 Konvertieren von string ... 117
2.3.4 Die Convert-Klasse ... 119
2.3.5 Die Parse-Methode ... 119
2.3.6 Boxing und Unboxing ... 120
2.4 Operatoren ... 121
2.4.1 Arithmetische Operatoren ... 122
2.4.2 Zuweisungsoperatoren ... 123
2.4.3 Logische Operatoren ... 124
2.4.4 Rangfolge der Operatoren ... 127
2.5 Kontrollstrukturen ... 128
2.5.1 Verzweigungsbefehle ... 128
2.5.2 Schleifenanweisungen ... 131

2.6 Benutzerdefinierte Datentypen ... 134
2.6.1 Enumerationen ... 134
2.6.2 Strukturen ... 135
2.7 Nutzerdefinierte Methoden ... 137
2.7.1 Methoden mit Rückgabewert ... 138
2.7.2 Methoden ohne Rückgabewert ... 139
2.7.3 Parameterübergabe mit ref ... 140
2.7.4 Parameterübergabe mit out ... 141
2.7.5 Methodenüberladung ... 142
2.7.6 Optionale Parameter ... 143
2.7.7 Benannte Parameter ... 144
2.8 Praxisbeispiele ... 145
2.8.1 Vom PAP zur Konsolenanwendung ... 145
2.8.2 Ein Konsolen- in ein Windows-Programm verwandeln ... 147
2.8.3 Schleifenanweisungen verstehen ... 149
2.8.4 Benutzerdefinierte Methoden überladen ... 151

3 OOP-Konzepte ... 155

3.1 Kleine Einführung in die OOP ... 155
3.1.1 Historische Entwicklung ... 155
3.1.2 Grundbegriffe der OOP ... 157
3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern ... 159
3.1.4 Allgemeiner Aufbau einer Klasse ... 160
3.1.5 Das Erzeugen eines Objekts ... 161
3.1.6 Einführungsbeispiel ... 164
3.2 Eigenschaften ... 169
3.2.1 Eigenschaften mit Zugriffsmethoden kapseln ... 169
3.2.2 Berechnete Eigenschaften ... 171
3.2.3 Lese-/Schreibschutz ... 173
3.2.4 Property-Accessoren ... 174
3.2.5 Statische Felder/Eigenschaften ... 174
3.2.6 Einfache Eigenschaften automatisch implementieren ... 177
3.3 Methoden ... 178
3.3.1 Öffentliche und private Methoden ... 178
3.3.2 Überladene Methoden ... 179
3.3.3 Statische Methoden ... 179

3.4 Ereignisse ... 181
3.4.1 Ereignis hinzufügen ... 182
3.4.2 Ereignis verwenden ... 185
3.5 Arbeiten mit Konstruktor und Destruktor ... 188
3.5.1 Konstruktor und Objektinitialisierer ... 188
3.5.2 Destruktor und Garbage Collector ... 191
3.5.3 Mit using den Lebenszyklus des Objekts kapseln ... 194
3.5.4 Verzögerte Initialisierung ... 194
3.6 Vererbung und Polymorphie ... 195
3.6.1 Klassendiagramm ... 195
3.6.2 Method-Overriding ... 197
3.6.3 Klassen implementieren ... 197
3.6.4 Implementieren der Objekte ... 200
3.6.5 Ausblenden von Mitgliedern durch Vererbung ... 202
3.6.6 Allgemeine Hinweise und Regeln zur Vererbung ... 204
3.6.7 Polymorphes Verhalten ... 205
3.6.8 Die Rolle von System.Object ... 208
3.7 Spezielle Klassen ... 209
3.7.1 Abstrakte Klassen ... 209
3.7.2 Versiegelte Klassen ... 210
3.7.3 Partielle Klassen ... 211
3.7.4 Statische Klassen ... 212
3.8 Schnittstellen (Interfaces) ... 213
3.8.1 Definition einer Schnittstelle ... 213
3.8.2 Implementieren einer Schnittstelle ... 214
3.8.3 Abfragen, ob Schnittstelle vorhanden ist ... 215
3.8.4 Mehrere Schnittstellen implementieren ... 215
3.8.5 Schnittstellenprogrammierung ist ein weites Feld ... 215
3.9 Praxisbeispiele ... 216
3.9.1 Eigenschaften sinnvoll kapseln ... 216
3.9.2 Eine statische Klasse anwenden ... 219
3.9.3 Vom fetten zum schlanken Client ... 221
3.9.4 Schnittstellenvererbung verstehen ... 232

4 Arrays, Strings, Funktionen ... 239

4.1 Datenfelder (Arrays) ... 239
4.1.1 Array deklarieren ... 239
4.1.2 Array instanziieren ... 240

4.1.3 Array initialisieren ... 240
4.1.4 Zugriff auf Array-Elemente ... 241
4.1.5 Zugriff mittels Schleife ... 242
4.1.6 Mehrdimensionale Arrays ... 243
4.1.7 Zuweisen von Arrays ... 245
4.1.8 Arrays aus Strukturvariablen ... 246
4.1.9 Löschen und Umdimensionieren von Arrays ... 247
4.1.10 Eigenschaften und Methoden von Arrays ... 248
4.1.11 Übergabe von Arrays ... 250
4.2 Verarbeiten von Zeichenketten ... 251
4.2.1 Zuweisen von Strings ... 251
4.2.2 Eigenschaften und Methoden von String-Variablen ... 252
4.2.3 Wichtige Methoden der String-Klasse ... 254
4.2.4 Die StringBuilder-Klasse ... 256
4.3 Reguläre Ausdrücke ... 258
4.3.1 Wozu werden reguläre Ausdrücke verwendet? ... 259
4.3.2 Eine kleine Einführung ... 259
4.3.3 Wichtige Methoden/Eigenschaften der Klasse Regex ... 260
4.3.4 Kompilierte reguläre Ausdrücke ... 262
4.3.5 RegexOptions-Enumeration ... 263
4.3.6 Metazeichen (Escape-Zeichen) ... 263
4.3.7 Zeichenmengen (Character Sets) ... 264
4.3.8 Quantifizierer ... 266
4.3.9 Zero-Width Assertions ... 267
4.3.10 Gruppen ... 270
4.3.11 Text ersetzen ... 271
4.3.12 Text splitten ... 272
4.4 Datums- und Zeitberechnungen ... 273
4.4.1 Die DateTime-Struktur ... 273
4.4.2 Wichtige Eigenschaften von DateTime-Variablen ... 274
4.4.3 Wichtige Methoden von DateTime-Variablen ... 275
4.4.4 Wichtige Mitglieder der DateTime-Struktur ... 275
4.4.5 Konvertieren von Datumstrings in DateTime-Werte ... 276
4.4.6 Die TimeSpan-Struktur ... 277
4.5 Mathematische Funktionen ... 278
4.5.1 Überblick ... 278
4.5.2 Zahlen runden ... 279
4.5.3 Winkel umrechnen ... 279

4.5.4 Potenz- und Wurzeloperationen ... 279
4.5.5 Logarithmus und Exponentialfunktionen ... 280
4.5.6 Zufallszahlen erzeugen ... 280
4.6 Zahlen- und Datumsformatierungen ... 281
4.6.1 Anwenden der ToString-Methode ... 281
4.6.2 Anwenden der Format-Methode ... 283
4.7 Praxisbeispiele ... 285
4.7.1 Zeichenketten verarbeiten ... 285
4.7.2 Zeichenketten mit StringBuilder addieren ... 288
4.7.3 Reguläre Ausdrücke testen ... 291
4.7.4 Methodenaufrufe mit Array-Parametern ... 293

5 Weitere Sprachfeatures ... 297

5.1 Namespaces (Namensräume) ... 297
5.1.1 Ein kleiner Überblick ... 297
5.1.2 Einen eigenen Namespace einrichten ... 298
5.1.3 Die using-Anweisung ... 299
5.1.4 Namespace Alias ... 300
5.1.5 Namespace Alias Qualifizierer ... 300
5.2 Operatorenüberladung ... 301
5.2.1 Syntaxregeln ... 301
5.2.2 Praktische Anwendung ... 302
5.3 Collections (Auflistungen) ... 303
5.3.1 Die Schnittstelle IEnumerable ... 303
5.3.2 ArrayList ... 305
5.3.3 Hashtable ... 307
5.3.4 Indexer ... 307
5.4 Generics ... 309
5.4.1 Klassische Vorgehensweise ... 310
5.4.2 Generics bieten Typsicherheit ... 311
5.4.3 Generische Methoden ... 312
5.4.4 Iteratoren ... 313
5.5 Generische Collections ... 314
5.5.1 List-Collection statt ArrayList ... 314
5.5.2 Vorteile generischer Collections ... 315
5.5.3 Constraints ... 315

5.6 Das Prinzip der Delegates ... 315
5.6.1 Delegates sind Methodenzeiger ... 316
5.6.2 Einen Delegate-Typ deklarieren ... 316
5.6.3 Ein Delegate-Objekt erzeugen ... 316
5.6.4 Delegates vereinfacht instanziieren ... 318
5.6.5 Anonyme Methoden ... 319
5.6.6 Lambda-Ausdrücke ... 320
5.6.7 Lambda-Ausdrücke in der Task Parallel Library ... 322
5.7 Dynamische Programmierung ... 324
5.7.1 Wozu dynamische Programmierung? ... 324
5.7.2 Das Prinzip der dynamischen Programmierung ... 324
5.7.3 Optionale Parameter sind hilfreich ... 327
5.7.4 Kovarianz und Kontravarianz ... 328
5.8 Weitere Datentypen ... 328
5.8.1 BigInteger ... 328
5.8.2 Complex ... 331
5.8.3 Tuple<> ... 331
5.8.4 SortedSet<> ... 332
5.9 Praxisbeispiele ... 333
5.9.1 ArrayList versus generische List ... 333
5.9.2 Generische IEnumerable-Interfaces implementieren ... 337
5.9.3 Delegates, anonyme Methoden, Lambda Expressions ... 340
5.9.4 Dynamischer Zugriff auf COM Interop ... 344

6 Einführung in LINQ ... 347

6.1 LINQ-Grundlagen ... 347
6.1.1 Die LINQ-Architektur ... 347
6.1.2 Anonyme Typen ... 348
6.1.3 Erweiterungsmethoden ... 350
6.2 Abfragen mit LINQ to Objects ... 351
6.2.1 Grundlegendes zur LINQ-Syntax ... 351
6.2.2 Zwei alternative Schreibweisen von LINQ Abfragen ... 352
6.2.3 Übersicht der wichtigsten Abfrage-Operatoren ... 353
6.3 LINQ-Abfragen im Detail ... 354
6.3.1 Die Projektionsoperatoren Select und SelectMany ... 355
6.3.2 Der Restriktionsoperator Where ... 356
6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy ... 357
6.3.4 Der Gruppierungsoperator GroupBy ... 358

6.3.5 Verknüpfen mit Join ... 361
6.3.6 Aggregat-Operatoren ... 361
6.3.7 Verzögertes Ausführen von LINQ-Abfragen ... 363
6.3.8 Konvertierungsmethoden ... 364
6.3.9 Abfragen mit PLINQ ... 364
6.4 Praxisbeispiele ... 367
6.4.1 Die Syntax von LINQ-Abfragen verstehen ... 367
6.4.2 Aggregat-Abfragen mit LINQ ... 370

Teil II: Technologien

7 Zugriff auf das Dateisystem ... 375
7.1 Grundlagen ... 375
7.1.1 Klassen für den Zugriff auf das Dateisystem ... 376
7.1.2 Statische versus Instanzen-Klasse ... 376
7.2 Übersichten ... 377
7.2.1 Methoden der Directory-Klasse ... 377
7.2.2 Methoden eines DirectoryInfo-Objekts ... 378
7.2.3 Eigenschaften eines DirectoryInfo-Objekts ... 378
7.2.4 Methoden der File-Klasse ... 378
7.2.5 Methoden eines FileInfo-Objekts ... 379
7.2.6 Eigenschaften eines FileInfo-Objekts ... 380
7.3 Operationen auf Verzeichnisebene ... 380
7.3.1 Existenz eines Verzeichnisses/einer Datei feststellen ... 380
7.3.2 Verzeichnisse erzeugen und löschen ... 381
7.3.3 Verzeichnisse verschieben und umbenennen ... 381
7.3.4 Aktuelles Verzeichnis bestimmen ... 382
7.3.5 Unterverzeichnisse ermitteln ... 382
7.3.6 Alle Laufwerke ermitteln ... 382
7.3.7 Dateien kopieren und verschieben ... 383
7.3.8 Dateien umbenennen ... 384
7.3.9 Dateiattribute feststellen ... 384
7.3.10 Verzeichnis einer Datei ermitteln ... 386
7.3.11 Alle im Verzeichnis enthaltenen Dateien ermitteln ... 386
7.3.12 Dateien und Unterverzeichnisse ermitteln ... 386

7.4 Zugriffsberechtigungen ... 387
7.4.1 ACL und ACE ... 387
7.4.2 SetAccessControl-Methode ... 388
7.4.3 Zugriffsrechte anzeigen ... 388
7.5 Weitere wichtige Klassen ... 389
7.5.1 Die Path-Klasse ... 389
7.5.2 Die Klasse FileSystemWatcher ... 390
7.6 Datei- und Verzeichnisdialoge ... 391
7.6.1 OpenFileDialog und SaveFileDialog ... 392
7.6.2 FolderBrowserDialog ... 393
7.7 Praxisbeispiele ... 394
7.7.1 Infos über Verzeichnisse und Dateien gewinnen ... 394
7.7.2 Eine Verzeichnisstruktur in die TreeView einlesen ... 398
7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen ... 400

8 Dateien lesen und schreiben ... 405

8.1 Grundprinzip der Datenpersistenz ... 405
8.1.1 Dateien und Streams ... 405
8.1.2 Die wichtigsten Klassen ... 406
8.1.3 Erzeugen eines Streams ... 407
8.2 Dateiparameter ... 407
8.2.1 FileAccess ... 407
8.2.2 FileMode ... 407
8.2.3 FileShare ... 408
8.3 Textdateien ... 408
8.3.1 Eine Textdatei beschreiben bzw. neu anlegen ... 408
8.3.2 Eine Textdatei lesen ... 410
8.4 Binärdateien ... 411
8.4.1 Lese-/Schreibzugriff ... 411
8.4.2 Die Methoden ReadAllBytes und WriteAllBytes ... 412
8.4.3 Erzeugen von BinaryReader/BinaryWriter ... 412
8.5 Sequenzielle Dateien ... 413
8.5.1 Lesen und schreiben von strukturierten Daten ... 413
8.5.2 Serialisieren von Objekten ... 414
8.6 Dateien verschlüsseln und komprimieren ... 415
8.6.1 Das Methodenpärchen Encrypt/Decrypt ... 415
8.6.2 Verschlüsseln unter Vista/Windows 7/Windows 8 ... 415

8.6.3 Verschlüsseln mit der CryptoStream-Klasse ... 416
8.6.4 Dateien komprimieren ... 417
8.7 Memory Mapped Files ... 418
8.7.1 Grundprinzip ... 418
8.7.2 Erzeugen eines MMF ... 419
8.7.3 Erstellen eines Map View ... 420
8.8 Praxisbeispiele ... 421
8.8.1 Auf eine Textdatei zugreifen ... 421
8.8.2 Einen Objektbaum persistent speichern ... 424
8.8.3 Ein Memory Mapped File (MMF) verwenden ... 431

9 Asynchrone Programmierung ... 435

9.1 Übersicht ... 435
9.1.1 Multitasking versus Multithreading ... 436
9.1.2 Deadlocks ... 437
9.1.3 Racing ... 437
9.2 Programmieren mit Threads ... 439
9.2.1 Einführungsbeispiel ... 439
9.2.2 Wichtige Thread-Methoden ... 440
9.2.3 Wichtige Thread-Eigenschaften ... 442
9.2.4 Einsatz der ThreadPool-Klasse ... 443
9.3 Sperrmechanismen ... 445
9.3.1 Threading ohne lock ... 445
9.3.2 Threading mit lock ... 447
9.3.3 Die Monitor-Klasse ... 449
9.3.4 Mutex ... 452
9.3.5 Methoden für die parallele Ausführung sperren ... 454
9.3.6 Semaphore ... 454
9.4 Interaktion mit der Programmoberfläche ... 456
9.4.1 Die Werkzeuge ... 456
9.4.2 Einzelne Steuerelemente mit Invoke aktualisieren ... 456
9.4.3 Mehrere Steuerelemente aktualisieren ... 458
9.4.4 Ist ein Invoke-Aufruf nötig? ... 458
9.4.5 Und was ist mit WPF? ... 459
9.5 Timer-Threads ... 460
9.6 Die BackgroundWorker-Komponente ... 461

9.7 Asynchrone Programmier-Entwurfsmuster ... 464
9.7.1 Kurzübersicht ... 464
9.7.2 Polling ... 465
9.7.3 Callback verwenden ... 467
9.7.4 Callback mit Parameterübergabe verwenden ... 468
9.7.5 Callback mit Zugriff auf die Programm-Oberfläche ... 469
9.8 Asynchroner Aufruf beliebiger Methoden ... 470
9.8.1 Die Beispielklasse ... 470
9.8.2 Asynchroner Aufruf ohne Callback ... 472
9.8.3 Asynchroner Aufruf mit Callback und Anzeigefunktion ... 473
9.8.4 Aufruf mit Rückgabewerten (per Eigenschaft) ... 474
9.8.5 Aufruf mit Rückgabewerten (per EndInvoke) ... 474
9.9 Es geht auch einfacher – async und await ... 475
9.9.1 Der Weg von Synchron zu Asynchron ... 476
9.9.2 Achtung: Fehlerquellen! ... 478
9.9.3 Eigene asynchrone Methoden entwickeln ... 480
9.10 Praxisbeispiele ... 482
9.10.1 Spieltrieb & Multithreading erleben ... 482
9.10.2 Prozess- und Thread-Informationen gewinnen ... 495
9.10.3 Ein externes Programm starten ... 500

10 Die Task Parallel Library ... 503

10.1 Überblick ... 503
10.1.1 Parallel-Programmierung ... 503
10.1.2 Möglichkeiten der TPL ... 506
10.1.3 Der CLR-Threadpool ... 506
10.2 Parallele Verarbeitung mit Parallel.Invoke ... 507
10.2.1 Aufrufvarianten ... 508
10.2.2 Einschränkungen ... 509
10.3 Verwendung von Parallel.For ... 509
10.3.1 Abbrechen der Verarbeitung ... 511
10.3.2 Auswerten des Verarbeitungsstatus ... 512
10.3.3 Und was ist mit anderen Iterator-Schrittweiten? ... 513
10.4 Collections mit Parallel.ForEach verarbeiten ... 513
10.5 Die Task-Klasse ... 514
10.5.1 Einen Task erzeugen ... 514
10.5.2 Den Task starten ... 515
10.5.3 Datenübergabe an den Task ... 517

10.5.4 Wie warte ich auf das Ende des Task? ... 518
10.5.5 Tasks mit Rückgabewerten ... 520
10.5.6 Die Verarbeitung abbrechen ... 523
10.5.7 Fehlerbehandlung ... 526
10.5.8 Weitere Eigenschaften ... 527
10.6 Zugriff auf das User-Interface ... 528
10.6.1 Task-Ende und Zugriff auf die Oberfläche ... 529
10.6.2 Zugriff auf das UI aus dem Task heraus ... 530
10.7 Weitere Datenstrukturen im Überblick ... 532
10.7.1 Threadsichere Collections ... 532
10.7.2 Primitive für die Threadsynchronisation ... 533
10.8 Parallel LINQ (PLINQ) ... 533
10.9 Die Parallel Diagnostic Tools ... 534
10.9.1 Fenster für parallele Aufgaben ... 534
10.9.2 Fenster für parallele Stacks ... 535
10.9.3 IntelliTrace ... 536
10.10 Praxisbeispiel: Spieltrieb – Version 2 ... 536
10.10.1 Aufgabenstellung ... 536
10.10.2 Global-Klasse ... 537
10.10.3 Controller-Klasse ... 538
10.10.4 LKW-Klasse ... 540
10.10.5 Schiff-Klasse ... 541
10.10.6 Oberfläche ... 544

11 Fehlersuche und Behandlung ... 547

11.1 Der Debugger ... 547
11.1.1 Allgemeine Beschreibung ... 547
11.1.2 Die wichtigsten Fenster ... 548
11.1.3 Debugging-Optionen ... 551
11.1.4 Praktisches Debugging am Beispiel ... 553
11.2 Arbeiten mit Debug und Trace ... 557
11.2.1 Wichtige Methoden von Debug und Trace ... 557
11.2.2 Besonderheiten der Trace-Klasse ... 561
11.2.3 TraceListener-Objekte ... 561
11.3 Caller Information ... 564
11.3.1 Attribute ... 564
11.3.2 Anwendung ... 564

11.4 Fehlerbehandlung ... 565
11.4.1 Anweisungen zur Fehlerbehandlung ... 565
11.4.2 try-catch ... 566
11.4.3 try-finally ... 571
11.4.4 Das Standardverhalten bei Ausnahmen festlegen ... 573
11.4.5 Die Exception-Klasse ... 574
11.4.6 Fehler/Ausnahmen auslösen ... 575
11.4.7 Eigene Fehlerklassen ... 575
11.4.8 Exceptionhandling zur Entwurfszeit ... 577
11.4.9 Code Contracts ... 578

Teil III: WPF-Anwendungen

12 Einführung in WPF ... 581

12.1 Neues aus der Gerüchteküche ... 582
12.1.1 Silverlight ... 582
12.1.2 WPF ... 582
12.2 Einführung ... 583
12.2.1 Was kann eine WPF-Anwendung? ... 583
12.2.2 Die eXtensible Application Markup Language ... 585
12.2.3 Verbinden von XAML und C#-Code ... 589
12.2.4 Zielplattformen ... 595
12.2.5 Applikationstypen ... 596
12.2.6 Vor- und Nachteile von WPF-Anwendungen ... 597
12.2.7 Weitere Dateien im Überblick ... 597
12.3 Alles beginnt mit dem Layout ... 600
12.3.1 Allgemeines zum Layout ... 600
12.3.2 Positionieren von Steuerelementen ... 602
12.3.3 Canvas ... 606
12.3.4 StackPanel ... 606
12.3.5 DockPanel ... 608
12.3.6 WrapPanel ... 610
12.3.7 UniformGrid ... 610
12.3.8 Grid ... 612
12.3.9 ViewBox ... 617
12.3.10 TextBlock ... 618

12.4 Das WPF-Programm ... 621
12.4.1 Die App-Klasse ... 621
12.4.2 Das Startobjekt festlegen ... 622
12.4.3 Kommandozeilenparameter verarbeiten ... 623
12.4.4 Die Anwendung beenden ... 624
12.4.5 Auswerten von Anwendungsereignissen ... 624
12.5 Die Window-Klasse ... 625
12.5.1 Position und Größe festlegen ... 625
12.5.2 Rahmen und Beschriftung ... 626
12.5.3 Das Fenster-Icon ändern ... 626
12.5.4 Anzeige weiterer Fenster ... 627
12.5.5 Transparenz ... 627
12.5.6 Abstand zum Inhalt festlegen ... 628
12.5.7 Fenster ohne Fokus anzeigen ... 628
12.5.8 Ereignisfolge bei Fenstern ... 629
12.5.9 Ein paar Worte zur Schriftdarstellung ... 629
12.5.10 Ein paar Worte zur Darstellung von Controls ... 632
12.5.11 Wird mein Fenster komplett mit WPF gerendert? ... 634

13 Übersicht WPF-Controls ... 635

13.1 Allgemeingültige Eigenschaften ... 635
13.2 Label ... 637
13.3 Button, RepeatButton, ToggleButton ... 638
13.3.1 Schaltflächen für modale Dialoge ... 638
13.3.2 Schaltflächen mit Grafik ... 639
13.4 TextBox, PasswordBox ... 640
13.4.1 TextBox ... 640
13.4.2 PasswordBox ... 642
13.5 CheckBox ... 643
13.6 RadioButton ... 645
13.7 ListBox, ComboBox ... 646
13.7.1 ListBox ... 646
13.7.2 ComboBox ... 649
13.7.3 Den Content formatieren ... 651
13.8 Image ... 652
13.8.1 Grafik per XAML zuweisen ... 652
13.8.2 Grafik zur Laufzeit zuweisen ... 653
13.8.3 Bild aus Datei laden ... 654

13.8.4 Die Grafikskalierung beeinflussen ... 655
13.9 MediaElement ... 656
13.10 Slider, ScrollBar ... 658
13.10.1 Slider ... 658
13.10.2 ScrollBar ... 659
13.11 ScrollViewer ... 660
13.12 Menu, ContextMenu ... 661
13.12.1 Menu ... 661
13.12.2 Tastenkürzel ... 662
13.12.3 Grafiken ... 663
13.12.4 Weitere Möglichkeiten ... 664
13.12.5 ContextMenu ... 665
13.13 ToolBar ... 666
13.14 StatusBar, ProgressBar ... 669
13.14.1 StatusBar ... 669
13.14.2 ProgressBar ... 671
13.15 Border, GroupBox, BulletDecorator ... 672
13.15.1 Border ... 672
13.15.2 GroupBox ... 673
13.15.3 BulletDecorator ... 674
13.16 RichTextBox ... 676
13.16.1 Verwendung und Anzeige von vordefiniertem Text ... 676
13.16.2 Neues Dokument zur Laufzeit erzeugen ... 678
13.16.3 Sichern von Dokumenten ... 678
13.16.4 Laden von Dokumenten ... 680
13.16.5 Texte per Code einfügen/modifizieren ... 681
13.16.6 Texte formatieren ... 682
13.16.7 EditingCommands ... 683
13.16.8 Grafiken/Objekte einfügen ... 684
13.16.9 Rechtschreibkontrolle ... 686
13.17 FlowDocumentPageViewer & Co. ... 686
13.17.1 FlowDocumentPageViewer ... 686
13.17.2 FlowDocumentReader ... 687
13.17.3 FlowDocumentScrollViewer ... 687
13.18 FlowDocument ... 687
13.18.1 FlowDocument per XAML beschreiben ... 688
13.18.2 FlowDocument per Code erstellen ... 690
13.19 DocumentViewer ... 691

13.20 Expander, TabControl ... 692
13.20.1 Expander ... 692
13.20.2 TabControl ... 694
13.21 Popup ... 695
13.22 TreeView ... 697
13.23 ListView ... 700
13.24 DataGrid ... 701
13.25 Calendar/DatePicker ... 702
13.26 InkCanvas ... 706
13.26.1 Stift-Parameter definieren ... 706
13.26.2 Die Zeichenmodi ... 707
13.26.3 Inhalte laden und sichern ... 708
13.26.4 Konvertieren in eine Bitmap ... 708
13.26.5 Weitere Eigenschaften ... 709
13.27 Ellipse, Rectangle, Line und Co. ... 710
13.27.1 Ellipse ... 710
13.27.2 Rectangle ... 710
13.27.3 Line ... 711
13.28 Browser ... 711
13.29 Ribbon ... 714
13.29.1 Allgemeine Grundlagen ... 714
13.29.2 Download/Installation ... 715
13.29.3 Erste Schritte ... 716
13.29.4 Registerkarten und Gruppen ... 717
13.29.5 Kontextabhängige Registerkarten ... 718
13.29.6 Einfache Beschriftungen ... 719
13.29.7 Schaltflächen ... 719
13.29.8 Auswahllisten ... 721
13.29.9 Optionsauswahl ... 724
13.29.10 Texteingaben ... 724
13.29.11 Screentips ... 724
13.29.12 Symbolleiste für den Schnellzugriff ... 725
13.29.13 Das RibbonWindow ... 726
13.29.14 Menüs ... 727
13.29.15 Anwendungsmenü ... 729
13.29.16 Alternativen ... 732
13.30 Chart ... 732
13.31 WindowsFormsHost ... 733

14 Wichtige WPF-Techniken ... 737

14.1 Eigenschaften ... 737
14.1.1 Abhängige Eigenschaften (Dependency Properties) ... 737
14.1.2 Angehängte Eigenschaften (Attached Properties) ... 739
14.2 Einsatz von Ressourcen ... 739
14.2.1 Was sind eigentlich Ressourcen? ... 739
14.2.2 Wo können Ressourcen gespeichert werden? ... 739
14.2.3 Wie definiere ich eine Ressource? ... 741
14.2.4 Statische und dynamische Ressourcen ... 742
14.2.5 Wie werden Ressourcen adressiert? ... 743
14.2.6 System-Ressourcen einbinden ... 744
14.3 Das WPF-Ereignis-Modell ... 744
14.3.1 Einführung ... 744
14.3.2 Routed Events ... 745
14.3.3 Direkte Events ... 747
14.4 Verwendung von Commands ... 748
14.4.1 Einführung zu Commands ... 748
14.4.2 Verwendung vordefinierter Commands ... 748
14.4.3 Das Ziel des Commands ... 750
14.4.4 Welche vordefinierten Commands stehen zur Verfügung? ... 751
14.4.5 Commands an Ereignismethoden binden ... 752
14.4.6 Wie kann ich ein Command per Code auslösen? ... 753
14.4.7 Command-Ausführung verhindern ... 754
14.5 Das WPF-Style-System ... 754
14.5.1 Übersicht ... 754
14.5.2 Benannte Styles ... 755
14.5.3 Typ-Styles ... 756
14.5.4 Styles anpassen und vererben ... 757
14.6 Verwenden von Triggern ... 760
14.6.1 Eigenschaften-Trigger (Property Triggers) ... 760
14.6.2 Ereignis-Trigger ... 762
14.6.3 Daten-Trigger ... 763
14.7 Einsatz von Templates ... 764
14.7.1 Neues Template erstellen ... 764
14.7.2 Template abrufen und verändern ... 768
14.8 Transformationen, Animationen, StoryBoards ... 771
14.8.1 Transformationen ... 771
14.8.2 Animationen mit dem StoryBoard realisieren ... 777

14.9 Praxisbeispiel ... 781
14.9.1 Arbeiten mit Microsoft Expression Blend ... 781

15 WPF-Datenbindung ... 787

15.1 Grundprinzip ... 787
15.1.1 Bindungsarten ... 788
15.1.2 Wann eigentlich wird die Quelle aktualisiert? ... 789
15.1.3 Geht es auch etwas langsamer? ... 790
15.1.4 Bindung zur Laufzeit realisieren ... 791
15.2 Binden an Objekte ... 793
15.2.1 Objekte im XAML-Code instanziieren ... 793
15.2.2 Verwenden der Instanz im C#-Quellcode ... 795
15.2.3 Anforderungen an die Quell-Klasse ... 795
15.2.4 Instanziieren von Objekten per C#-Code ... 797
15.3 Binden von Collections ... 798
15.3.1 Anforderung an die Collection ... 798
15.3.2 Einfache Anzeige ... 799
15.3.3 Navigieren zwischen den Objekten ... 800
15.3.4 Einfache Anzeige in einer ListBox ... 802
15.3.5 DataTemplates zur Anzeigeformatierung ... 803
15.3.6 Mehr zu List- und ComboBox ... 804
15.3.7 Verwendung der ListView ... 806
15.4 Noch einmal zurück zu den Details ... 808
15.4.1 Navigieren in den Daten ... 808
15.4.2 Sortieren ... 810
15.4.3 Filtern ... 810
15.4.4 Live Shaping ... 811
15.5 Anzeige von Datenbankinhalten ... 812
15.5.1 Datenmodell per LINQ to SQL-Designer erzeugen ... 813
15.5.2 Die Programm-Oberfläche ... 814
15.5.3 Der Zugriff auf die Daten ... 815
15.6 Drag & Drop-Datenbindung ... 816
15.6.1 Vorgehensweise ... 816
15.6.2 Weitere Möglichkeiten ... 819
15.7 Formatieren von Werten ... 820
15.7.1 IValueConverter ... 821
15.7.2 BindingBase.StringFormat-Eigenschaft ... 823
15.8 Das DataGrid als Universalwerkzeug ... 825

15.8.1 Grundlagen der Anzeige ... 825
15.8.2 UI-Virtualisierung ... 826
15.8.3 Spalten selbst definieren ... 826
15.8.4 Zusatzinformationen in den Zeilen anzeigen ... 828
15.8.5 Vom Betrachten zum Editieren ... 829
15.9 Praxisbeispiele ... 830
15.9.1 Collections in Hintergrundthreads füllen ... 830
15.9.2 Drag & Drop-Bindung bei 1:n-Beziehungen ... 833

16 Druckausgabe mit WPF ... 839

16.1 Grundlagen ... 839
16.1.1 XPS-Dokumente ... 839
16.1.2 System.Printing ... 840
16.1.3 System.Windows.Xps ... 841
16.2 Einfache Druckausgaben mit dem PrintDialog ... 841
16.3 Mehrseitige Druckvorschau-Funktion ... 844
16.3.1 Fix-Dokumente ... 844
16.3.2 Flow-Dokumente ... 850
16.4 Druckerinfos, -auswahl, -konfiguration ... 853
16.4.1 Die installierten Drucker bestimmen ... 854
16.4.2 Den Standarddrucker bestimmen ... 855
16.4.3 Mehr über einzelne Drucker erfahren ... 855
16.4.4 Spezifische Druckeinstellungen vornehmen ... 857
16.4.5 Direkte Druckausgabe ... 859

Teil IV: Windows Store Apps

17 Erste Schritte in WinRT ... 863

17.1 Grundkonzepte und Begriffe ... 863
17.1.1 Windows Runtime (WinRT) ... 863
17.1.2 Windows Store Apps ... 864
17.1.3 Fast and Fluid ... 865
17.1.4 Process Sandboxing und Contracts ... 866
17.1.5 .NET WinRT-Profil ... 868
17.1.6 Language Projection ... 868
17.1.7 Vollbildmodus ... 870

17.1.8 Windows Store ... 870
17.1.9 Zielplattformen ... 871
17.2 Entwurfsumgebung ... 872
17.2.1 Betriebssystem ... 872
17.2.2 Windows-Simulator ... 873
17.2.3 Remote-Debugging ... 875
17.3 Ein (kleines) Einstiegsbeispiel ... 876
17.3.1 Aufgabenstellung ... 876
17.3.2 Quellcode ... 876
17.3.3 Oberflächenentwurf ... 879
17.3.4 Installation und Test ... 881
17.3.5 Verbesserungen ... 882
17.3.6 Fazit ... 885
17.4 Weitere Details zu WinRT ... 887
17.4.1 Wo ist WinRT einzuordnen? ... 887
17.4.2 Die WinRT-API ... 888
17.4.3 Wichtige WinRT-Namespaces ... 890
17.4.4 Der Unterbau ... 891
17.5 Gedanken zum Thema "WinRT & Tablets" ... 894
17.5.1 Windows 8-Oberfläche versus Desktop ... 894
17.5.2 Tablets und Touchscreens ... 894
17.6 Praxisbeispiel ... 896
17.6.1 WinRT in Desktop-Applikationen nutzen ... 896

18 WinRT-Oberflächen entwerfen ... 899

18.1 Grundkonzepte ... 899
18.1.1 XAML (oder HTML 5) für die Oberfläche ... 900
18.1.2 Die Page, der Frame und das Window ... 901
18.1.3 Das Befehlsdesign ... 902
18.1.4 Die Navigationsdesigns ... 904
18.1.5 Achtung: Fingereingabe! ... 905
18.1.6 Verwendung von Schriftarten ... 906
18.2 Projekttypen und Seitentemplates ... 906
18.2.1 Leere App ... 906
18.2.2 Geteilte App (Split App) ... 908
18.2.3 Raster-App (Grid App) ... 910
18.2.4 Leere Seite (Blank Page) ... 911

18.2.5 Standardseite (Basic Page) ... 912
18.2.6 Ein eigenes Grundlayout erstellen ... 914
18.3 Seitenauswahl und -navigation ... 915
18.3.1 Die Startseite festlegen ... 915
18.3.2 Navigation und Parameterübergabe ... 915
18.3.3 Den Seitenstatus erhalten ... 916
18.4 Die vier App-Ansichten ... 917
18.4.1 Vollbild quer und hochkant ... 917
18.4.2 Angedockt und Füllmodus ... 918
18.4.3 Reagieren auf die Änderung ... 918
18.4.4 Angedockten Modus aktiv beenden ... 922
18.5 Skalieren von Apps ... 922
18.6 Praxisbeispiele ... 924
18.6.1 Seitennavigation und Parameterübergabe ... 924
18.6.2 Auf Ansichtsänderungen reagieren ... 926
18.7 Tipps & Tricks ... 930
18.7.1 Symbole für WinRT-Oberflächen finden ... 930
18.7.2 Wie werde ich das Grufti-Layout schnell los? ... 931

19 Die wichtigsten Controls ... 933

19.1 Einfache WinRT-Controls ... 933
19.1.1 TextBlock, RichTextBlock ... 933
19.1.2 Button, HyperlinkButton, RepeatButton ... 936
19.1.3 CheckBox, RadioButton, ToggleButton, ToggleSwitch ... 938
19.1.4 TextBox, PasswordBox, RichEditBox ... 939
19.1.5 Image ... 943
19.1.6 ScrollBar, Slider, ProgressBar, ProgressRing ... 945
19.1.7 Border, Ellipse, Rectangle ... 946
19.2 Layout-Controls ... 947
19.2.1 Canvas ... 947
19.2.2 StackPanel ... 948
19.2.3 ScrollViewer ... 948
19.2.4 Grid ... 949
19.2.5 VariableSizedWrapGrid ... 950
19.3 Listendarstellungen ... 951
19.3.1 ComboBox, ListBox ... 951
19.3.2 ListView ... 955

19.3.3 GridView ... 956
19.3.4 FlipView ... 959
19.4 Sonstige Controls ... 961
19.4.1 CaptureElement ... 961
19.4.2 MediaElement ... 962
19.4.3 Frame ... 964
19.4.4 WebView ... 964
19.4.5 ToolTip ... 965
19.5 Praxisbeispiele ... 967
19.5.1 Einen StringFormat-Konverter implementieren ... 967
19.5.2 Besonderheiten der TextBox kennen lernen ... 969
19.5.3 Daten in der GridView gruppieren ... 972
19.5.4 Das SemanticZoom-Control verwenden ... 977
19.5.5 Die CollectionViewSource verwenden ... 981
19.5.6 Zusammenspiel ListBox/AppBar ... 985
19.5.7 Musikwiedergabe im Hintergrund realisieren ... 988

20 Apps im Detail ... 993

20.1 Ein Windows Store App-Projekt im Detail ... 993
20.1.1 Contracts und Extensions ... 994
20.1.2 AssemblyInfo.cs ... 995
20.1.3 Verweise ... 996
20.1.4 App.xaml und App.xaml.cs ... 997
20.1.5 Package.appxmanifest ... 998
20.1.6 Application1_TemporaryKey.pfx ... 1002
20.1.7 MainPage.xaml & MainPage.xaml.cs ... 1002
20.1.8 Datentyp-Konverter/Hilfsklassen ... 1003
20.1.9 StandardStyles.xaml ... 1005
20.1.10 Assets/Symbole ... 1006
20.1.11 Nach dem Kompilieren ... 1007
20.2 Der Lebenszyklus einer WinRT-App ... 1007
20.2.1 Möglichkeiten der Aktivierung von Apps ... 1009
20.2.2 Der Splash Screen ... 1011
20.2.3 Suspending ... 1011
20.2.4 Resuming ... 1012
20.2.5 Beenden von Apps ... 1013
20.2.6 Die Ausnahmen von der Regel ... 1014
20.2.7 Debuggen ... 1014

20.3 Daten speichern und laden ... 1018
20.3.1 Grundsätzliche Überlegungen ... 1018
20.3.2 Worauf und wie kann ich zugreifen? ... 1018
20.3.3 Das AppData-Verzeichnis ... 1018
20.3.4 Das Anwendungs-Installationsverzeichnis ... 1021
20.3.5 Das Downloads-Verzeichnis ... 1022
20.3.6 Sonstige Verzeichnisse ... 1023
20.3.7 Anwendungsdaten lokal sichern und laden ... 1023
20.3.8 Daten in der Cloud ablegen/laden (Roaming) ... 1025
20.3.9 Aufräumen ... 1028
20.3.10 Sensible Informationen speichern ... 1028
20.4 Praxisbeispiele ... 1030
20.4.1 Unterstützung für den Search-Contract bieten ... 1030
20.4.2 Die Auto-Play-Funktion unterstützen ... 1037
20.4.3 Einen zusätzlichen Splash Screen einsetzen ... 1041
20.4.4 Eine Dateiverknüpfung erstellen ... 1043

21 WinRT-Techniken ... 1049

21.1 Arbeiten mit Dateien/Verzeichnissen ... 1049
21.1.1 Verzeichnisinformationen auflisten ... 1049
21.1.2 Unterverzeichnisse auflisten ... 1052
21.1.3 Verzeichnisse erstellen/löschen ... 1053
21.1.4 Dateien auflisten ... 1055
21.1.5 Dateien erstellen/schreiben/lesen ... 1057
21.1.6 Dateien kopieren/umbenennen/löschen ... 1061
21.1.7 Verwenden der Dateipicker ... 1062
21.1.8 StorageFile-/StorageFolder-Objekte speichern ... 1069
21.1.9 Verwenden der Most Recently Used-Liste ... 1071
21.2 Datenaustausch zwischen Apps/Programmen ... 1072
21.2.1 Zwischenablage ... 1073
21.2.2 Teilen von Inhalten ... 1080
21.2.3 Eine App als Freigabeziel verwenden ... 1083
21.2.4 Zugriff auf die Kontaktliste ... 1084
21.3 Spezielle Oberflächenelemente ... 1086
21.3.1 MessageDialog ... 1086
21.3.2 Popup-Benachrichtigungen ... 1089
21.3.3 PopUp/Flyouts ... 1097

21.3.4 Das PopupMenu einsetzen ... 1101
21.3.5 Eine AppBar verwenden ... 1103
21.4 Datenbanken und Windows Store Apps ... 1108
21.4.1 Der Retter in der Not: SQLite! ... 1108
21.4.2 Verwendung/Kurzüberblick ... 1108
21.4.3 Installation ... 1110
21.4.4 Wie kommen wir zu einer neuen Datenbank? ... 1112
21.4.5 Wie werden die Daten manipuliert? ... 1116
21.5 Vertrieb der App ... 1118
21.5.1 Verpacken der App ... 1118
21.5.2 Windows App Certification Kit ... 1120
21.5.3 App-Installation per Skript ... 1122
21.6 Ein Blick auf die App-Schwachstellen ... 1123
21.6.1 Quellcodes im Installationsverzeichnis ... 1123
21.6.2 Zugriff auf den App-Datenordner ... 1125
21.7 Praxisbeispiele ... 1125
21.7.1 Ein Verzeichnis auf Änderungen überwachen ... 1125
21.7.2 Eine App als Freigabeziel verwenden ... 1128
21.7.3 ToastNotifications einfach erzeugen ... 1133

Anhang

A Glossar ... 1141

B Wichtige Dateiextensions ... 1147

Index ... 1149

Bonuskapitel im E-Book

Zweites Vorwort **1193**

Teil V: Weitere Technologien

22 XML in Theorie und Praxis **1197**

22.1 XML – etwas Theorie 1197
22.1.1 Übersicht 1197
22.1.2 Der XML-Grundaufbau 1200
22.1.3 Wohlgeformte Dokumente 1201
22.1.4 Processing Instructions (PI) 1204
22.1.5 Elemente und Attribute 1204
22.1.6 Verwendbare Zeichensätze 1206
22.2 XSD-Schemas 1208
22.2.1 XSD-Schemas und ADO.NET 1208
22.2.2 XML-Schemas in Visual Studio analysieren 1210
22.2.3 XML-Datei mit XSD-Schema erzeugen 1213
22.2.4 XSD-Schema aus einer XML-Datei erzeugen 1214
22.3 Verwendung des DOM unter .NET 1215
22.3.1 Übersicht 1215
22.3.2 DOM-Integration in C# 1216
22.3.3 Laden von Dokumenten 1216
22.3.4 Erzeugen von XML-Dokumenten 1217
22.3.5 Auslesen von XML-Dateien 1219
22.3.6 Direktzugriff auf einzelne Elemente 1221
22.3.7 Einfügen von Informationen 1221
22.3.8 Suchen in den Baumzweigen 1224
22.4 XML-Verarbeitung mit LINQ to XML 1227
22.4.1 Die LINQ to XML-API 1227
22.4.2 Neue XML-Dokumente erzeugen 1229

– Diese Kapitel finden Sie nur im E-Book –

22.4.3 Laden und Sichern von XML-Dokumenten ... 1231
22.4.4 Navigieren in XML-Daten ... 1232
22.4.5 Auswählen und Filtern ... 1234
22.4.6 Manipulieren der XML-Daten ... 1234
22.4.7 XML-Dokumente transformieren ... 1236
22.5 Weitere Möglichkeiten der XML-Verarbeitung ... 1239
22.5.1 Die relationale Sicht mit XmlDataDocument ... 1239
22.5.2 XML-Daten aus Objektstrukturen erzeugen ... 1242
22.5.3 Schnelles Suchen in XML-Daten mit XPathNavigator ... 1245
22.5.4 Schnelles Auslesen von XML-Daten mit XmlReader ... 1248
22.5.5 Erzeugen von XML-Daten mit XmlWriter ... 1250
22.5.6 XML transformieren mit XSLT ... 1252
22.6 Praxisbeispiele ... 1254
22.6.1 Mit dem DOM in XML-Dokumenten navigieren ... 1254
22.6.2 XML-Daten in eine TreeView einlesen ... 1257

23 Einführung in ADO.NET ... 1261

23.1 Eine kleine Übersicht ... 1261
23.1.1 Die ADO.NET-Klassenhierarchie ... 1261
23.1.2 Die Klassen der Datenprovider ... 1262
23.1.3 Das Zusammenspiel der ADO.NET-Klassen ... 1265
23.2 Das Connection-Objekt ... 1266
23.2.1 Allgemeiner Aufbau ... 1266
23.2.2 OleDbConnection ... 1266
23.2.3 Schließen einer Verbindung ... 1268
23.2.4 Eigenschaften des Connection-Objekts ... 1268
23.2.5 Methoden des Connection-Objekts ... 1270
23.2.6 Der ConnectionStringBuilder ... 1271
23.3 Das Command-Objekt ... 1271
23.3.1 Erzeugen und Anwenden eines Command-Objekts ... 1272
23.3.2 Erzeugen mittels CreateCommand-Methode ... 1272
23.3.3 Eigenschaften des Command-Objekts ... 1273
23.3.4 Methoden des Command-Objekts ... 1275
23.3.5 Freigabe von Connection- und Command-Objekten ... 1276
23.4 Parameter-Objekte ... 1277
23.4.1 Erzeugen und Anwenden eines Parameter-Objekts ... 1277
23.4.2 Eigenschaften des Parameter-Objekts ... 1278

– Diese Kapitel finden Sie nur im E-Book –

23.5 Das CommandBuilder-Objekt ... 1279
23.5.1 Erzeugen ... 1279
23.5.2 Anwenden ... 1279
23.6 Das DataReader-Objekt ... 1280
23.6.1 DataReader erzeugen ... 1280
23.6.2 Daten lesen ... 1281
23.6.3 Eigenschaften des DataReaders ... 1282
23.6.4 Methoden des DataReaders ... 1282
23.7 Das DataAdapter-Objekt ... 1283
23.7.1 DataAdapter erzeugen ... 1283
23.7.2 Command-Eigenschaften ... 1284
23.7.3 Fill-Methode ... 1285
23.7.4 Update-Methode ... 1286
23.8 Praxisbeispiele ... 1287
23.8.1 Wichtige ADO.NET-Objekte im Einsatz ... 1287
23.8.2 Eine Aktionsabfrage ausführen ... 1288
23.8.3 Eine Auswahlabfrage aufrufen ... 1291
23.8.4 Die Datenbank aktualisieren ... 1293

24 Das DataSet ... 1297

24.1 Grundlegende Features des DataSets ... 1297
24.1.1 Die Objekthierarchie ... 1298
24.1.2 Die wichtigsten Klassen ... 1298
24.1.3 Erzeugen eines DataSets ... 1299
24.2 Das DataTable-Objekt ... 1300
24.2.1 DataTable erzeugen ... 1301
24.2.2 Spalten hinzufügen ... 1301
24.2.3 Zeilen zur DataTable hinzufügen ... 1302
24.2.4 Auf den Inhalt einer DataTable zugreifen ... 1303
24.3 Die DataView ... 1305
24.3.1 Erzeugen einer DataView ... 1305
24.3.2 Sortieren und Filtern von Datensätzen ... 1305
24.3.3 Suchen von Datensätzen ... 1306
24.4 Typisierte DataSets ... 1306
24.4.1 Ein typisiertes DataSet erzeugen ... 1307
24.4.2 Das Konzept der Datenquellen ... 1308
24.4.3 Typisierte DataSets und TableAdapter ... 1309

– Diese Kapitel finden Sie nur im E-Book –

24.5 Die Qual der Wahl ... 1310
24.5.1 DataReader – der schnelle Lesezugriff ... 1311
24.5.2 DataSet – die Datenbank im Hauptspeicher ... 1311
24.5.3 Objektrelationales Mapping – die Zukunft? ... 1312
24.6 Praxisbeispiele ... 1313
24.6.1 In der DataView sortieren und filtern ... 1313
24.6.2 Suche nach Datensätzen ... 1315
24.6.3 Ein DataSet in einen XML-String serialisieren ... 1316
24.6.4 Untypisiertes in ein typisiertes DataSet konvertieren ... 1321
24.6.5 Eine LINQ to SQL-Abfrage ausführen ... 1326

25 OOP-Spezial ... 1331

25.1 Eine kleine Einführung in die UML ... 1331
25.1.1 Use Case-Diagramm ... 1331
25.1.2 Use Case-Dokumentation ... 1333
25.1.3 Objekte identifizieren ... 1334
25.1.4 Statisches Modell ... 1335
25.1.5 Beziehungen zwischen den Klassen ... 1336
25.1.6 Dynamisches Modell ... 1336
25.1.7 Implementierung ... 1337
25.1.8 Test-Client ... 1341
25.2 Der Klassen-Designer ... 1344
25.2.1 Ein neues Klassendiagramm erzeugen ... 1345
25.2.2 Werkzeugkasten ... 1346
25.2.3 Enumeration ... 1347
25.2.4 Klasse ... 1349
25.2.5 Struktur ... 1351
25.2.6 Abstrakte Klasse ... 1351
25.2.7 Schnittstelle ... 1353
25.2.8 Delegate ... 1355
25.2.9 Zuordnung ... 1357
25.2.10 Vererbung ... 1358
25.2.11 Diagramme anpassen ... 1358
25.2.12 Wann lohnt sich der Klassen-Designer? ... 1359
25.3 Praxisbeispiele ... 1359
25.3.1 Implementierung einer Finite State Machine ... 1359
25.3.2 Modellierung des Bestellsystems einer Firma ... 1365

– Diese Kapitel finden Sie nur im E-Book –

26 Das Microsoft Event Pattern ... 1379

26.1 Einführung in Design Pattern ... 1379
26.2 Aufbau und Bedeutung des Observer Pattern ... 1380
26.2.1 Subjekt und Observer ... 1380
26.2.2 Sequenzdiagramme ... 1382
26.2.3 Die Registration-Sequenz ... 1382
26.2.4 Die Notification-Sequenz ... 1383
26.2.5 Die Unregistration-Sequenz ... 1383
26.2.6 Bedeutung der Sequenzen für das Geschäftsmodell ... 1384
26.2.7 Die Rolle des Containers ... 1384
26.3 Implementieren mit Interfaces und Callbacks ... 1385
26.3.1 Übersicht und Klassendiagramm ... 1385
26.3.2 Die Schnittstellen IObserver und IObservable ... 1387
26.3.3 Die Abstrakte Klasse Subject ... 1387
26.3.4 Observer1 ... 1388
26.3.5 Observer2 ... 1389
26.3.6 Model ... 1390
26.3.7 Form1 ... 1391
26.3.8 Ein zweites Klassendiagramm ... 1392
26.3.9 Testen ... 1393
26.4 Implementierung mit Delegates und Events ... 1394
26.4.1 Multicast-Events ... 1395
26.4.2 IObserver, IObservable und Subject ... 1395
26.4.3 Observer1 und Observer2 ... 1396
26.4.4 Model ... 1396
26.4.5 Form1 ... 1397
26.4.6 Test und Vergleich ... 1397
26.4.7 Klassendiagramm ... 1398
26.5 Implementierung des Microsoft Event Pattern ... 1399
26.5.1 Namensgebung für Ereignisse ... 1399
26.5.2 Namensgebung und Signatur der Delegates ... 1399
26.5.3 Hinzufügen einer das Ereignis auslösenden Methode ... 1400
26.5.4 Model ... 1401
26.5.5 Observer1 ... 1402
26.5.6 Form1 ... 1403
26.6 Test und Vergleich ... 1403
26.7 Klassendiagramm ... 1403

– Diese Kapitel finden Sie nur im E-Book –

26.8 Implementierung eines Event Pattern 1404
26.9 Praxisbeispiel 1406
26.9.1 Objekte beobachten sich gegenseitig 1406

27 Verteilen von Anwendungen 1417

27.1 ClickOnce-Deployment 1418
27.1.1 Übersicht/Einschränkungen 1418
27.1.2 Die Vorgehensweise 1419
27.1.3 Ort der Veröffentlichung 1419
27.1.4 Anwendungsdateien 1420
27.1.5 Erforderliche Komponenten 1420
27.1.6 Aktualisierungen 1421
27.1.7 Veröffentlichungsoptionen 1422
27.1.8 Veröffentlichen 1423
27.1.9 Verzeichnisstruktur 1423
27.1.10 Der Webpublishing-Assistent 1425
27.1.11 Neue Versionen erstellen 1426
27.2 InstallShield 1426
27.2.1 Installation 1426
27.2.2 Aktivieren 1427
27.2.3 Ein neues Setup-Projekt 1427
27.2.4 Finaler Test 1435

28 Weitere Techniken 1437

28.1 Zugriff auf die Zwischenablage 1437
28.1.1 Das Clipboard-Objekt 1437
28.1.2 Zwischenablage-Funktionen für Textboxen 1439
28.2 Arbeiten mit der Registry 1439
28.2.1 Allgemeines 1440
28.2.2 Registry-Unterstützung in .NET 1442
28.3 .NET-Reflection 1443
28.3.1 Übersicht 1443
28.3.2 Assembly laden 1443
28.3.3 Mittels GetType und Type Informationen sammeln 1444
28.3.4 Dynamisches Laden von Assemblies 1446
28.4 Das SerialPort-Control 1449
28.4.1 Übersicht 1449
28.4.2 Einführungsbeispiele 1450

– Diese Kapitel finden Sie nur im E-Book –

28.4.3 Thread-Probleme bei Windows Forms-Anwendungen ... 1453
28.4.4 Ein einfaches Terminalprogramm ... 1456
28.5 Praxisbeispiele ... 1461
28.5.1 Zugriff auf die Registry ... 1461
28.5.2 Dateiverknüpfungen erzeugen ... 1463
28.5.3 Betrachter für Manifestressourcen ... 1465
28.5.4 Ressourcen mit Reflection auslesen ... 1468

29 Konsolenanwendungen ... 1471

29.1 Grundaufbau/Konzepte ... 1471
29.1.1 Unser Hauptprogramm – Program.cs ... 1472
29.1.2 Rückgabe eines Fehlerstatus ... 1473
29.1.3 Parameterübergabe ... 1474
29.1.4 Zugriff auf die Umgebungsvariablen ... 1475
29.2 Die Kommandozentrale: System.Console ... 1476
29.2.1 Eigenschaften ... 1477
29.2.2 Methoden/Ereignisse ... 1477
29.2.3 Textausgaben ... 1478
29.2.4 Farbangaben ... 1479
29.2.5 Tastaturabfragen ... 1480
29.2.6 Arbeiten mit Streamdaten ... 1481
29.3 Praxisbeispiel ... 1483
29.3.1 Farbige Konsolenanwendung ... 1483
29.3.2 Weitere Hinweise und Beispiele ... 1485

Teil VI: Windows Forms

30 Windows Forms-Anwendungen ... 1489

30.1 Grundaufbau/Konzepte ... 1489
30.1.1 Das Hauptprogramm – Program.cs ... 1490
30.1.2 Die Oberflächendefinition – Form1.Designer.cs ... 1494
30.1.3 Die Spielwiese des Programmierers – Form1.cs ... 1495
30.1.4 Die Datei AssemblyInfo.cs ... 1496
30.1.5 Resources resx/Resources.Designer.cs ... 1497
30.1.6 Settings.settings/Settings.Designer.cs ... 1498
30.1.7 Settings.cs ... 1499

– Diese Kapitel finden Sie nur im E-Book –

30.2 Ein Blick auf die Application-Klasse ... 1500
30.2.1 Eigenschaften ... 1500
30.2.2 Methoden ... 1501
30.2.3 Ereignisse ... 1503
30.3 Allgemeine Eigenschaften von Komponenten ... 1503
30.3.1 Font ... 1504
30.3.2 Handle ... 1506
30.3.3 Tag ... 1507
30.3.4 Modifiers ... 1507
30.4 Allgemeine Ereignisse von Komponenten ... 1508
30.4.1 Die Eventhandler-Argumente ... 1508
30.4.2 Sender ... 1508
30.4.3 Der Parameter e ... 1510
30.4.4 Mausereignisse ... 1510
30.4.5 KeyPreview ... 1512
30.4.6 Weitere Ereignisse ... 1513
30.4.7 Validätsprüfungen ... 1513
30.4.8 SendKeys ... 1514
30.5 Allgemeine Methoden von Komponenten ... 1515

31 Windows Forms-Formulare ... 1517

31.1 Übersicht ... 1517
31.1.1 Wichtige Eigenschaften des Form-Objekts ... 1518
31.1.2 Wichtige Ereignisse des Form-Objekts ... 1520
31.1.3 Wichtige Methoden des Form-Objekts ... 1521
31.2 Praktische Aufgabenstellungen ... 1522
31.2.1 Fenster anzeigen ... 1522
31.2.2 Splash Screens beim Anwendungsstart anzeigen ... 1525
31.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen ... 1528
31.2.4 Ein Formular durchsichtig machen ... 1528
31.2.5 Die Tabulatorreihenfolge festlegen ... 1529
31.2.6 Ausrichten von Komponenten im Formular ... 1530
31.2.7 Spezielle Panels für flexible Layouts ... 1532
31.2.8 Menüs erzeugen ... 1534
31.3 MDI-Anwendungen ... 1538
31.3.1 "Falsche" MDI-Fenster bzw. Verwenden von Parent ... 1538
31.3.2 Die echten MDI-Fenster ... 1539
31.3.3 Die Kindfenster ... 1540

– Diese Kapitel finden Sie nur im E-Book –

31.3.4 Automatisches Anordnen der Kindfenster ... 1541
31.3.5 Zugriff auf die geöffneten MDI-Kindfenster ... 1542
31.3.6 Zugriff auf das aktive MDI-Kindfenster ... 1543
31.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü ... 1543
31.4 Praxisbeispiele ... 1544
31.4.1 Informationsaustausch zwischen Formularen ... 1544
31.4.2 Ereigniskette beim Laden/Entladen eines Formulars ... 1551

32 Windows Forms-Komponenten ... 1557

32.1 Allgemeine Hinweise ... 1557
32.1.1 Hinzufügen von Komponenten ... 1557
32.1.2 Komponenten zur Laufzeit per Code erzeugen ... 1558
32.2 Allgemeine Steuerelemente ... 1560
32.2.1 Label ... 1560
32.2.2 LinkLabel ... 1561
32.2.3 Button ... 1562
32.2.4 TextBox ... 1563
32.2.5 MaskedTextBox ... 1566
32.2.6 CheckBox ... 1567
32.2.7 RadioButton ... 1569
32.2.8 ListBox ... 1569
32.2.9 CheckedListBox ... 1571
32.2.10 ComboBox ... 1571
32.2.11 PictureBox ... 1572
32.2.12 DateTimePicker ... 1573
32.2.13 MonthCalendar ... 1573
32.2.14 HScrollBar, VScrollBar ... 1574
32.2.15 TrackBar ... 1575
32.2.16 NumericUpDown ... 1575
32.2.17 DomainUpDown ... 1576
32.2.18 ProgressBar ... 1576
32.2.19 RichTextBox ... 1577
32.2.20 ListView ... 1578
32.2.21 TreeView ... 1584
32.2.22 WebBrowser ... 1589
32.3 Container ... 1590
32.3.1 FlowLayout/TableLayout/SplitContainer ... 1590
32.3.2 Panel ... 1590

– Diese Kapitel finden Sie nur im E-Book –

32.3.3 GroupBox ... 1591
32.3.4 TabControl ... 1592
32.3.5 ImageList ... 1594
32.4 Menüs & Symbolleisten ... 1595
32.4.1 MenuStrip und ContextMenuStrip ... 1595
32.4.2 ToolStrip ... 1595
32.4.3 StatusStrip ... 1595
32.4.4 ToolStripContainer ... 1596
32.5 Daten ... 1596
32.5.1 DataSet ... 1596
32.5.2 DataGridView/DataGrid ... 1597
32.5.3 BindingNavigator/BindingSource ... 1597
32.5.4 Chart ... 1597
32.6 Komponenten ... 1599
32.6.1 ErrorProvider ... 1599
32.6.2 HelpProvider ... 1599
32.6.3 ToolTip ... 1599
32.6.4 BackgroundWorker ... 1599
32.6.5 Timer ... 1599
32.6.6 SerialPort ... 1600
32.7 Drucken ... 1600
32.7.1 PrintPreviewControl ... 1600
32.7.2 PrintDocument ... 1600
32.8 Dialoge ... 1600
32.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog ... 1600
32.8.2 FontDialog/ColorDialog ... 1601
32.9 WPF-Unterstützung mit dem ElementHost ... 1601
32.10 Praxisbeispiele ... 1601
32.10.1 Mit der CheckBox arbeiten ... 1601
32.10.2 Steuerelemente per Code selbst erzeugen ... 1603
32.10.3 Controls-Auflistung im TreeView anzeigen ... 1605
32.10.4 WPF-Komponenten mit dem ElementHost anzeigen ... 1609

33 Grundlagen Grafikausgabe ... 1613

33.1 Übersicht und erste Schritte ... 1613
33.1.1 GDI+ – Ein erster Einblick für Umsteiger ... 1614
33.1.2 Namespaces für die Grafikausgabe ... 1615

– Diese Kapitel finden Sie nur im E-Book –

33.2 Darstellen von Grafiken ... 1617
33.2.1 Die PictureBox-Komponente ... 1617
33.2.2 Das Image-Objekt ... 1618
33.2.3 Laden von Grafiken zur Laufzeit ... 1619
33.2.4 Sichern von Grafiken ... 1619
33.2.5 Grafikeigenschaften ermitteln ... 1620
33.2.6 Erzeugen von Vorschaugrafiken (Thumbnails) ... 1621
33.2.7 Die Methode RotateFlip ... 1622
33.2.8 Skalieren von Grafiken ... 1623
33.3 Das .NET-Koordinatensystem ... 1624
33.3.1 Globale Koordinaten ... 1625
33.3.2 Seitenkoordinaten (globale Transformation) ... 1626
33.3.3 Gerätekoordinaten (Seitentransformation) ... 1628
33.4 Grundlegende Zeichenfunktionen von GDI+ ... 1629
33.4.1 Das zentrale Graphics-Objekt ... 1629
33.4.2 Punkte zeichnen/abfragen ... 1632
33.4.3 Linien ... 1633
33.4.4 Kantenglättung mit Antialiasing ... 1634
33.4.5 PolyLine ... 1635
33.4.6 Rechtecke ... 1635
33.4.7 Polygone ... 1637
33.4.8 Splines ... 1638
33.4.9 Bézierkurven ... 1639
33.4.10 Kreise und Ellipsen ... 1640
33.4.11 Tortenstück (Segment) ... 1640
33.4.12 Bogenstück ... 1642
33.4.13 Wo sind die Rechtecke mit den runden Ecken? ... 1643
33.4.14 Textausgabe ... 1644
33.4.15 Ausgabe von Grafiken ... 1648
33.5 Unser Werkzeugkasten ... 1649
33.5.1 Einfache Objekte ... 1649
33.5.2 Vordefinierte Objekte ... 1651
33.5.3 Farben/Transparenz ... 1653
33.5.4 Stifte (Pen) ... 1654
33.5.5 Pinsel (Brush) ... 1657
33.5.6 SolidBrush ... 1658
33.5.7 HatchBrush ... 1658
33.5.8 TextureBrush ... 1659

– Diese Kapitel finden Sie nur im E-Book –

33.5.9 LinearGradientBrush ... 1660
33.5.10 PathGradientBrush ... 1661
33.5.11 Fonts ... 1662
33.5.12 Path-Objekt ... 1663
33.5.13 Clipping/Region ... 1666
33.6 Standarddialoge ... 1670
33.6.1 Schriftauswahl ... 1670
33.6.2 Farbauswahl ... 1671
33.7 Praxisbeispiele ... 1673
33.7.1 Ein Graphics-Objekt erzeugen ... 1673
33.7.2 Zeichenoperationen mit der Maus realisieren ... 1675

34 Druckausgabe ... 1679

34.1 Einstieg und Übersicht ... 1679
34.1.1 Nichts geht über ein Beispiel ... 1679
34.1.2 Programmiermodell ... 1681
34.1.3 Kurzübersicht der Objekte ... 1682
34.2 Auswerten der Druckereinstellungen ... 1682
34.2.1 Die vorhandenen Drucker ... 1682
34.2.2 Der Standarddrucker ... 1683
34.2.3 Verfügbare Papierformate/Seitenabmessungen ... 1684
34.2.4 Der eigentliche Druckbereich ... 1685
34.2.5 Die Seitenausrichtung ermitteln ... 1686
34.2.6 Ermitteln der Farbfähigkeit ... 1686
34.2.7 Die Druckauflösung abfragen ... 1686
34.2.8 Ist beidseitiger Druck möglich? ... 1687
34.2.9 Einen "Informationsgerätekontext" erzeugen ... 1687
34.2.10 Abfragen von Werten während des Drucks ... 1688
34.3 Festlegen von Druckereinstellungen ... 1689
34.3.1 Einen Drucker auswählen ... 1689
34.3.2 Drucken in Millimetern ... 1689
34.3.3 Festlegen der Seitenränder ... 1690
34.3.4 Druckjobname ... 1691
34.3.5 Anzahl der Kopien ... 1692
34.3.6 Beidseitiger Druck ... 1692
34.3.7 Seitenzahlen festlegen ... 1693
34.3.8 Druckqualität verändern ... 1696
34.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen ... 1697

– Diese Kapitel finden Sie nur im E-Book –

34.4 Die Druckdialoge verwenden ... 1697
34.4.1 PrintDialog ... 1698
34.4.2 PageSetupDialog ... 1699
34.4.3 PrintPreviewDialog ... 1701
34.4.4 Ein eigenes Druckvorschau-Fenster realisieren ... 1702
34.5 Drucken mit OLE-Automation ... 1703
34.5.1 Kurzeinstieg in die OLE-Automation ... 1703
34.5.2 Drucken mit Microsoft Word ... 1705
34.6 Praxisbeispiele ... 1707
34.6.1 Den Drucker umfassend konfigurieren ... 1707
34.6.2 Diagramme mit dem Chart-Control drucken ... 1717
34.6.3 Druckausgabe mit Word ... 1719

35 Windows Forms-Datenbindung ... 1725

35.1 Prinzipielle Möglichkeiten ... 1725
35.2 Manuelle Bindung an einfache Datenfelder ... 1726
35.2.1 BindingSource erzeugen ... 1726
35.2.2 Binding-Objekt ... 1727
35.2.3 DataBindings-Collection ... 1727
35.3 Manuelle Bindung an Listen und Tabellen ... 1727
35.3.1 DataGridView ... 1728
35.3.2 Datenbindung von ComboBox und ListBox ... 1728
35.4 Entwurfszeit-Bindung an typisierte DataSets ... 1728
35.5 Drag & Drop-Datenbindung ... 1730
35.6 Navigations- und Bearbeitungsfunktionen ... 1730
35.6.1 Navigieren zwischen den Datensätzen ... 1730
35.6.2 Hinzufügen und Löschen ... 1730
35.6.3 Aktualisieren und Abbrechen ... 1731
35.6.4 Verwendung des BindingNavigators ... 1731
35.7 Die Anzeigedaten formatieren ... 1732
35.8 Praxisbeispiele ... 1732
35.8.1 Einrichten und Verwenden einer Datenquelle ... 1732
35.8.2 Eine Auswahlabfrage im DataGridView anzeigen ... 1736
35.8.3 Master-Detailbeziehungen im DataGrid anzeigen ... 1739
35.8.4 Datenbindung Chart-Control ... 1740

– Diese Kapitel finden Sie nur im E-Book –

36 Erweiterte Grafikausgabe 1745

36.1 Transformieren mit der Matrix-Klasse 1745
36.1.1 Übersicht 1745
36.1.2 Translation 1746
36.1.3 Skalierung 1746
36.1.4 Rotation 1747
36.1.5 Scherung 1747
36.1.6 Zuweisen der Matrix 1748
36.2 Low-Level-Grafikmanipulationen 1748
36.2.1 Worauf zeigt Scan0? 1749
36.2.2 Anzahl der Spalten bestimmen 1750
36.2.3 Anzahl der Zeilen bestimmen 1751
36.2.4 Zugriff im Detail (erster Versuch) 1751
36.2.5 Zugriff im Detail (zweiter Versuch) 1753
36.2.6 Invertieren 1755
36.2.7 In Graustufen umwandeln 1756
36.2.8 Heller/Dunkler 1757
36.2.9 Kontrast 1759
36.2.10 Gamma-Wert 1760
36.2.11 Histogramm spreizen 1760
36.2.12 Ein universeller Grafikfilter 1763
36.3 Fortgeschrittene Techniken 1767
36.3.1 Flackerfrei dank Double Buffering 1767
36.3.2 Animationen 1769
36.3.3 Animated GIFs 1772
36.3.4 Auf einzelne GIF-Frames zugreifen 1774
36.3.5 Transparenz realisieren 1776
36.3.6 Eine Grafik maskieren 1777
36.3.7 JPEG-Qualität beim Sichern bestimmen 1779
36.4 Grundlagen der 3D-Vektorgrafik 1780
36.4.1 Datentypen für die Verwaltung 1780
36.4.2 Eine universelle 3D-Grafik-Klasse 1781
36.4.3 Grundlegende Betrachtungen 1782
36.4.4 Translation 1785
36.4.5 Streckung/Skalierung 1786
36.4.6 Rotation 1787
36.4.7 Die eigentlichen Zeichenroutinen 1789

– Diese Kapitel finden Sie nur im E-Book –

36.5 Und doch wieder GDI-Funktionen ... 1791
36.5.1 Am Anfang war das Handle ... 1791
36.5.2 Gerätekontext (Device Context Types) ... 1794
36.5.3 Koordinatensysteme und Abbildungsmodi ... 1796
36.5.4 Zeichenwerkzeuge/Objekte ... 1800
36.5.5 Bitmaps ... 1802
36.6 Praxisbeispiele ... 1806
36.6.1 Die Transformationsmatrix verstehen ... 1806
36.6.2 Eine 3D-Grafikausgabe in Aktion ... 1809
36.6.3 Einen Fenster-Screenshot erzeugen ... 1812

37 Ressourcen/Lokalisierung ... 1815

37.1 Manifestressourcen ... 1815
37.1.1 Erstellen von Manifestressourcen ... 1815
37.1.2 Zugriff auf Manifestressourcen ... 1817
37.2 Typisierte Ressourcen ... 1819
37.2.1 Erzeugen von resources-Dateien ... 1819
37.2.2 Hinzufügen der resources-Datei zum Projekt ... 1819
37.2.3 Zugriff auf die Inhalte von resources-Dateien ... 1820
37.2.4 ResourceManager einer resources-Datei erzeugen ... 1820
37.2.5 Was sind resx-Dateien? ... 1821
37.3 Streng typisierte Ressourcen ... 1821
37.3.1 Erzeugen streng typisierter Ressourcen ... 1822
37.3.2 Verwenden streng typisierter Ressourcen ... 1822
37.3.3 Streng typisierte Ressourcen per Reflection auslesen ... 1823
37.4 Anwendungen lokalisieren ... 1825
37.4.1 Localizable und Language ... 1825
37.4.2 Beispiel "Landesfahnen" ... 1825
37.4.3 Einstellen der aktuellen Kultur zur Laufzeit ... 1828

38 Komponentenentwicklung ... 1831

38.1 Überblick ... 1831
38.2 Benutzersteuerelement ... 1832
38.2.1 Entwickeln einer Auswahl-ListBox ... 1832
38.2.2 Komponente verwenden ... 1834
38.3 Benutzerdefiniertes Steuerelement ... 1835
38.3.1 Entwickeln eines BlinkLabels ... 1835
38.3.2 Verwenden der Komponente ... 1838

– Diese Kapitel finden Sie nur im E-Book –

38.4 Komponentenklasse ... 1838
38.5 Eigenschaften ... 1839
38.5.1 Einfache Eigenschaften ... 1839
38.5.2 Schreib-/Lesezugriff (Get/Set) ... 1839
38.5.3 Nur Lese-Eigenschaft (ReadOnly) ... 1840
38.5.4 Nur-Schreibzugriff (WriteOnly) ... 1841
38.5.5 Hinzufügen von Beschreibungen ... 1841
38.5.6 Ausblenden im Eigenschaftenfenster ... 1841
38.5.7 Einfügen in Kategorien ... 1842
38.5.8 Default-Wert einstellen ... 1842
38.5.9 Standard-Eigenschaft (Indexer) ... 1843
38.5.10 Wertebereichsbeschränkung und Fehlerprüfung ... 1843
38.5.11 Eigenschaften von Aufzählungstypen ... 1845
38.5.12 Standard Objekt-Eigenschaften ... 1846
38.5.13 Eigene Objekt-Eigenschaften ... 1846
38.6 Methoden ... 1848
38.6.1 Konstruktor ... 1849
38.6.2 Class-Konstruktor ... 1850
38.6.3 Destruktor ... 1851
38.6.4 Aufruf des Basisklassen-Konstruktors ... 1851
38.6.5 Aufruf von Basisklassen-Methoden ... 1852
38.7 Ereignisse (Events) ... 1852
38.7.1 Ereignis mit Standardargument definieren ... 1853
38.7.2 Ereignis mit eigenen Argumenten ... 1854
38.7.3 Ein Default-Ereignis festlegen ... 1855
38.7.4 Mit Ereignissen auf Windows-Messages reagieren ... 1855
38.8 Weitere Themen ... 1857
38.8.1 Wohin mit der Komponente? ... 1857
38.8.2 Assembly-Informationen festlegen ... 1858
38.8.3 Assemblies signieren ... 1861
38.8.4 Komponenten-Ressourcen einbetten ... 1861
38.8.5 Der Komponente ein Icon zuordnen ... 1862
38.8.6 Den Designmodus erkennen ... 1863
38.8.7 Komponenten lizenzieren ... 1863
38.9 Praxisbeispiele ... 1867
38.9.1 AnimGif – Anzeige von Animationen ... 1867
38.9.2 Eine FontComboBox entwickeln ... 1870
38.9.3 Das PropertyGrid verwenden ... 1872

– Diese Kapitel finden Sie nur im E-Book –

Teil VII: ASP.NET

39 Einführung in ASP.NET ... 1877

39.1 ASP.NET für Ein- und Umsteiger ... 1877
39.1.1 ASP – Ein kurzer Blick zurück ... 1877
39.1.2 Was ist bei ASP.NET anders? ... 1878
39.1.3 Was gibt es noch in ASP.NET? ... 1880
39.1.4 Vorteile von ASP.NET gegenüber ASP ... 1881
39.1.5 Voraussetzungen für den Einsatz von ASP.NET ... 1882
39.1.6 Und was hat das alles mit C# zu tun? ... 1882
39.2 Unsere erste Web-Anwendung ... 1885
39.2.1 Visueller Entwurf der Bedienoberfläche ... 1885
39.2.2 Zuweisen der Objekteigenschaften ... 1888
39.2.3 Verknüpfen der Objekte mit Ereignissen ... 1889
39.2.4 Programm kompilieren und testen ... 1890
39.3 Die ASP.NET-Projektdateien ... 1891
39.3.1 Die ASP.NET-Projekttypen ... 1892
39.3.2 ASPX-Datei(en) ... 1894
39.3.3 Die aspx.cs-Datei(en) ... 1896
39.3.4 Die Datei Global.asax ... 1897
39.3.5 Das Startformular ... 1898
39.3.6 Die Datei Web.config ... 1898
39.3.7 Masterpages (master-Dateien) ... 1901
39.3.8 Sitemap (Web.sitemap) ... 1901
39.3.9 Benutzersteuerelemente (ascx-Dateien) ... 1902
39.3.10 Die Web-Projekt-Verzeichnisse ... 1902
39.4 Lernen am Beispiel ... 1903
39.4.1 Erstellen des Projekts ... 1903
39.4.2 Oberflächengestaltung ... 1904
39.4.3 Ereignisprogrammierung ... 1905
39.4.4 Ein Fehler, was nun? ... 1907
39.4.5 Ereignisse von Textboxen ... 1908
39.4.6 Ein gemeinsamer Ereignis-Handler ... 1909
39.4.7 Eingabefokus setzen ... 1909
39.4.8 Ausgaben in einer Tabelle ... 1910
39.4.9 Scrollen der Anzeige ... 1912
39.4.10 Zusammenspiel mehrerer Formulare ... 1913
39.4.11 Umleiten bei Direktaufruf ... 1914

– Diese Kapitel finden Sie nur im E-Book –

39.4.12 Ärger mit den Cookies ... 1915
39.4.13 Export auf den IIS ... 1917
39.5 Tipps & Tricks ... 1918
39.5.1 Nachinstallieren IIS 7 bzw. 7.5 (Windows 7) ... 1918
39.5.2 Nachinstallieren IIS8 (Windows 8) ... 1919

40 Übersicht ASP.NET-Controls ... 1921

40.1 Einfache Steuerelemente im Überblick ... 1921
40.1.1 Label ... 1921
40.1.2 TextBox ... 1923
40.1.3 Button, ImageButton, LinkButton ... 1924
40.1.4 CheckBox, RadioButton ... 1925
40.1.5 CheckBoxList, BulletList, RadioButtonList ... 1926
40.1.6 Table ... 1927
40.1.7 Hyperlink ... 1929
40.1.8 Image, ImageMap ... 1929
40.1.9 Calendar ... 1931
40.1.10 Panel ... 1932
40.1.11 HiddenField ... 1932
40.1.12 Substitution ... 1933
40.1.13 XML ... 1934
40.1.14 FileUpload ... 1936
40.1.15 AdRotator ... 1937
40.2 Steuerelemente für die Seitennavigation ... 1938
40.2.1 Mehr Übersicht mit Web.Sitemap ... 1938
40.2.2 Menu ... 1940
40.2.3 TreeView ... 1943
40.2.4 SiteMapPath ... 1946
40.2.5 MultiView, View ... 1947
40.2.6 Wizard ... 1948
40.3 Webseitenlayout/-design ... 1950
40.3.1 Masterpages ... 1950
40.3.2 Themes/Skins ... 1953
40.3.3 Webparts ... 1956
40.4 Die Validator-Controls ... 1957
40.4.1 Übersicht ... 1957
40.4.2 Wo findet die Fehlerprüfung statt? ... 1958
40.4.3 Verwendung ... 1958

– Diese Kapitel finden Sie nur im E-Book –

40.4.4 RequiredFieldValidator ... 1959
40.4.5 CompareValidator ... 1960
40.4.6 RangeValidator ... 1962
40.4.7 RegularExpressionValidator ... 1962
40.4.8 CustomValidator ... 1963
40.4.9 ValidationSummary ... 1966
40.4.10 Weitere Möglichkeiten ... 1967
40.5 Praxisbeispiele ... 1967
40.5.1 Themes und Skins verstehen ... 1967
40.5.2 Masterpages verwenden ... 1972
40.5.3 Webparts verwenden ... 1975

41 Datenbindung unter ASP.NET ... 1981

41.1 Einstiegsbeispiel ... 1981
41.1.1 Erstellen der ASP.NET-Website ... 1981
41.2 Einführung ... 1986
41.2.1 Konzept ... 1986
41.2.2 Übersicht über die DataSource-Steuerelemente ... 1987
41.3 SQLDataSource ... 1988
41.3.1 Datenauswahl mit Parametern ... 1990
41.3.2 Parameter für INSERT, UPDATE und DELETE ... 1991
41.3.3 Methoden ... 1993
41.3.4 Caching ... 1994
41.3.5 Aktualisieren/Refresh ... 1995
41.4 AccessDataSource ... 1995
41.5 ObjectDataSource ... 1995
41.5.1 Verbindung zwischen Objekt und DataSource ... 1995
41.5.2 Ein Beispiel sorgt für Klarheit ... 1997
41.5.3 Geschäftsobjekte in einer Session verwalten ... 2001
41.6 SitemapDataSource ... 2003
41.7 LinqDataSource ... 2004
41.7.1 Bindung von einfachen Collections ... 2004
41.7.2 Bindung eines LINQ to SQL-DataContext ... 2005
41.8 EntityDataSource ... 2007
41.8.1 Entity Data Model erstellen ... 2007
41.8.2 EntityDataSource anbinden ... 2010
41.8.3 Datenmenge filtern ... 2013
41.9 XmlDataSource ... 2013

– Diese Kapitel finden Sie nur im E-Book –

41.10 QueryExtender ... 2015
41.10.1 Grundlagen ... 2015
41.10.2 Suchen ... 2016
41.10.3 Sortieren ... 2018
41.11 GridView ... 2019
41.11.1 Auswahlfunktion (Zeilenauswahl) ... 2019
41.11.2 Auswahl mit mehrspaltigem Index ... 2020
41.11.3 Hyperlink-Spalte für Detailansicht ... 2020
41.11.4 Spalten erzeugen ... 2021
41.11.5 Paging realisieren ... 2022
41.11.6 Edit, Update, Delete ... 2024
41.11.7 Keine Daten, was tun? ... 2024
41.12 DetailsView ... 2024
41.13 FormView ... 2026
41.14 DataList ... 2029
41.14.1 Bearbeitungsfunktionen implementieren ... 2030
41.14.2 Layout verändern ... 2031
41.15 Repeater ... 2032
41.16 ListView ... 2033
41.17 Typisierte Datenbindung ... 2033
41.18 Model Binding ... 2034
41.19 Chart ... 2036

42 ASP.NET-Objekte/-Techniken ... 2039

42.1 Wichtige ASP.NET-Objekte ... 2039
42.1.1 HTTPApplication ... 2039
42.1.2 Application ... 2042
42.1.3 Session ... 2043
42.1.4 Page ... 2045
42.1.5 Request ... 2048
42.1.6 Response ... 2051
42.1.7 Server ... 2055
42.1.8 Cookies verwenden ... 2056
42.2 Fehlerbehandlung unter ASP.NET ... 2059
42.2.1 Fehler beim Entwurf ... 2059
42.2.2 Laufzeitfehler ... 2059
42.2.3 Eine eigene Fehlerseite ... 2061
42.2.4 Fehlerbehandlung im Web Form ... 2062

– Diese Kapitel finden Sie nur im E-Book –

42.2.5 Fehlerbehandlung in der Anwendung ... 2063
42.2.6 Alternative Fehlerseite einblenden ... 2064
42.2.7 Lokale Fehlerbehandlung ... 2065
42.2.8 Seite nicht gefunden – was nun? ... 2066
42.3 E-Mail-Versand in ASP.NET ... 2066
42.3.1 Übersicht ... 2067
42.3.2 Mail-Server bestimmen ... 2067
42.3.3 Einfache Text-E-Mails versenden ... 2069
42.3.4 E-Mails mit Dateianhang ... 2070
42.4 Sicherheit von Webanwendungen ... 2071
42.4.1 Authentication ... 2071
42.4.2 Forms Authentication realisieren ... 2072
42.4.3 Impersonation ... 2076
42.4.4 Authorization ... 2077
42.4.5 Administrieren der Website ... 2079
42.4.6 Steuerelemente für das Login-Handling ... 2083
42.4.7 Programmieren der Sicherheitseinstellungen ... 2087
42.5 AJAX in ASP.NET-Anwendungen ... 2089
42.5.1 Was ist AJAX und was kann es? ... 2089
42.5.2 Die AJAX-Controls ... 2090
42.5.3 AJAX-Control-Toolkit ... 2094
42.6 User Controls/Webbenutzersteuerelemente ... 2095
42.6.1 Ein simples Einstiegsbeispiel ... 2096
42.6.2 Dynamische Grafiken im User Control anzeigen ... 2099
42.6.3 Grafikausgaben per User Control realisieren ... 2104

Teil VIII: Silverlight

43 Silverlight-Entwicklung ... 2109
43.1 Einführung ... 2109
43.1.1 Zielplattformen ... 2110
43.1.2 Silverlight-Applikationstypen ... 2110
43.1.3 Wichtige Unterschiede zu den WPF-Anwendungen ... 2112
43.1.4 Vor- und Nachteile von Silverlight-Anwendungen ... 2114
43.1.5 Entwicklungstools ... 2116
43.1.6 Installation auf dem Client ... 2116

– Diese Kapitel finden Sie nur im E-Book –

43.2 Die Silverlight-Anwendung im Detail ... 2117
43.2.1 Ein kleines Beispielprojekt ... 2118
43.2.2 Das Application Package und das Test-Web ... 2120
43.3 Die Projektdateien im Überblick ... 2123
43.3.1 Projektverwaltung mit App.xaml & App.xaml.cs ... 2124
43.3.2 MainPage.xaml & MainPage.xaml.cs ... 2126
43.3.3 AssemblyInfo.cs ... 2127
43.4 Fenster und Seiten in Silverlight ... 2127
43.4.1 Das Standardfenster ... 2128
43.4.2 Untergeordnete Silverlight-Fenster ... 2129
43.4.3 UserControls für die Anzeige von Detaildaten ... 2131
43.4.4 Echte Windows ... 2132
43.4.5 Navigieren in Silverlight-Anwendungen ... 2133
43.5 Datenbanken/Datenbindung ... 2138
43.5.1 ASP.NET-Webdienste/WCF-Dienste ... 2139
43.5.2 WCF Data Services ... 2148
43.6 Isolierter Speicher ... 2159
43.6.1 Grundkonzept ... 2159
43.6.2 Das virtuelle Dateisystem verwalten ... 2160
43.6.3 Arbeiten mit Dateien ... 2163
43.7 Fulltrust-Anwendungen ... 2164
43.8 Praxisbeispiele ... 2167
43.8.1 Eine Out-of-Browser-Applikation realisieren ... 2167
43.8.2 Out-of-Browser-Anwendung aktualisieren ... 2171
43.8.3 Testen auf aktive Internetverbindung ... 2172
43.8.4 Auf Out-of-Browser-Anwendung testen ... 2173
43.8.5 Den Browser bestimmen ... 2173
43.8.6 Parameter an das Plug-in übergeben ... 2174
43.8.7 Auf den QueryString zugreifen ... 2176
43.8.8 Timer in Silverlight nutzen ... 2177
43.8.9 Dateien lokal speichern ... 2178
43.8.10 Drag & Drop realisieren ... 2180
43.8.11 Auf die Zwischenablage zugreifen ... 2181
43.8.12 Weitere Fenster öffnen ... 2183

Index ... 2187

– Diese Kapitel finden Sie nur im E-Book –

Vorwort

C# ist eine noch junge Sprache, sie bietet Ihnen die Möglichkeiten und Flexibilität von C++ und erlaubt trotzdem eine schnelle und unkomplizierte Programmierpraxis wie Visual Basic. C# ist (fast) genauso mächtig wie C++, wurde aber komplett neu auf objektorientierter Basis geschrieben.

Damit ist C# das ideale Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework, beginnend bei Windows Forms- über WPF-, ASP.NET- , WinRT- und Silverlight-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch ist ein faires Angebot für künftige wie auch für fortgeschrittene C#-Programmierer. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen zwanzig Jahren zu verschiedenen Programmiersprachen geschrieben haben:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie C# in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip "so viel wie nötig" sich lediglich eine "Initialisierungsfunktion" auf die Fahnen schreiben kann.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt unser Titel für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual C# 2012 zu liefern oder all die Informationen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip "so viel wie nötig" eine schmale Schneise durch den Urwald der .NET-Programmierung mit Visual C# 2012 schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.

- Für den *Profi* wollen wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Da mit Visual C# 2012 zahlreiche neue Features (vor allem die WinRT-Programmierung) neu hinzugekommen sind und wir auch viele Leserwünsche zusätzlich eingearbeitet haben, mussten einige Kapitel wesentlich erweitert bzw. neu hinzugefügt werden.

Ein Vergleich der Inhaltsverzeichnisse zeigt, dass aus den ursprünglichen 40 Kapiteln nunmehr 43 Kapitel geworden sind, die inzwischen 2200 Seiten umfassen.

Die Kapitel des Buchs haben wir in acht Themenkomplexen gruppiert:

- Grundlagen der Programmierung mit Visual C# 2012
- Technologien der Programmentwicklung
- WPF-Anwendungen
- Windows Store Apps
- Weitere Technologien
- Windows Forms
- ASP.NET
- Silverlight

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Im gedruckten Teil dieses Buchs finden Sie die ersten vier Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen vier Themenkomplexe mussten wir in ein E-Book auslagern, welches Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Neuigkeiten in Visual Studio 2012

Mit dem Erscheinen von Windows 8 bietet Microsoft erstmals ein Tablet-taugliches Betriebssystem an, für das Sie als C#-Entwickler eigene Anwendungen (neudeutsch Apps) entwickeln können. Während es für iPad und Android-Geräte mittlerweile viele tausende Apps gibt, herrscht bei Windows 8 noch ein riesengroßer Nachholbedarf.

Dieses Buch soll Ihnen helfen, die dafür nötigen Grundkenntnisse zu gewinnen und eigene Apps zu entwickeln. Dabei helfen Ihnen neben den fünf WinRT-Kapiteln auch die WPF-Kapitel über die entsprechenden Basistechnologien (XAML/Datenbindung).

Leider ist nicht alles Gold was glänzt, und so hat Microsoft mit der Visual Studio 2012, dem .NET-Framework 4.5 und Windows 8 zwar vieles anders, aber nicht alles besser gemacht. Wer seinen Blick durch diverse Foren streifen lässt, der wird schnell den Unmut über viele Entscheidungen bei den Entwicklern spüren:

- Das freudlose "Gruftilayout" der neuen Visual Studio-IDE und die Beschriftung des Hauptmenüs in Großbuchstaben sind nicht jedermanns Geschmack.
- Das Setup-Template wurde aus Visual Studio entfernt, Sie dürfen *InstallShield Limited Edition* verwenden (Download per Website).
- Das .NET-Framework 4.5 wird nicht mehr unter Windows XP unterstützt. Der XP-Marktanteil lag im September 2012 noch bei 42%, genauso hoch wie bei Windows 7. Auf diese Kundengruppe werden viele Entwickler nicht verzichten wollen. Damit aber bleibt nur das "alte" Framework 4.0 als Zielframework für neue Projekte.
- Windows 8-Projekte (WinRT) können nur unter Windows 8 erstellt werden und im Wesentlichen auch nur per Microsoft Store vertrieben werden.

Wir hoffen, dass der Leser dieses Buchs obige kritische Worte höher zu schätzen weiß als den euphorischen Lobgesang manch anderer Autoren auf jedwede Art von "Highlights".

Weitere Bücher von uns

Auf drei weitere von uns verfasste Buchtitel, die sich ebenfalls auf Visual Studio 2012 beziehen, wollen wir Sie hier noch hinweisen:

- Eine ideale Ergänzung zum vorliegenden Buch ist unser "Visual C# 2012 – Kochbuch". Mit mehr als 500 How-to-Problemlösungen zu allen hier behandelten Grundlagenthemen sind Sie bestens für die Anforderungen der Praxis gewappnet und können weitere Lücken schließen.
- Das Pendant zum vorliegenden Buch ist unser im gleichen Verlag erschienener Buchtitel "Visual Basic 2012 – Grundlagen und Profiwissen". Da es das gleiche Inhaltsverzeichnis hat (inklusive Beispielcode), lassen sich ideale Vergleiche zwischen beiden Sprachen anstellen. Eine solche "Übersetzungshilfe" scheint besonders wichtig zu sein, weil einerseits viele altgediente Visual Basic-Programmierer zu C# wechseln werden und man andererseits in einem .NET-Entwicklerteam durchaus in mehreren .NET-Sprachen zusammenarbeitet.
- Der Datenbank- und Web-Programmierung widmet sich ausführlich unser bei Microsoft Press erschienener Spezialtitel "Datenbankprogrammierung mit Visual C# 2012".

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs können Sie sich unter der Adresse

LINK: `http://www.doko-buch.de`

herunterladen.

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z.B. mittels F5-Taste kompilieren und starten können.

- Einige wenige Datenbankprojekte verwenden absolute Pfadnamen, die Sie vor dem Kompilieren des Beispiels erst noch anpassen müssen.
- Für einige Beispiele sind ein installierter Microsoft SQL Server Express LocalDB sowie der Microsoft Internet Information Server (ASP.NET) erforderlich.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio 2012 unter Windows 8 ausführen.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

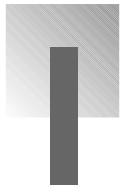
Sie werden – trotz der rund 2200 Seiten – in diesem Buch nicht alles finden, was Visual C# 2012 bzw. das .NET Framework 4.5 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen optimalen und überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gern über unsere Website kontaktieren:

LINK: `http://www.doko-buch.de`

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Walter Doberenz und Thomas Gewinnus *Wintersdorf/Frankfurt/O., im Oktober 2012*



Teil I: Grundlagen

- **Einstieg in Visual Studio 2012**
- **Grundlagen der Sprache C#**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2012

Dieses Kapitel bietet Ihnen einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2012. Gleich nachdem Sie die Hürden der Installation gemeistert haben, erstellen Sie Ihre ersten .NET-Anwendungen, werden dabei en passant mit den grundlegenden Features der Entwicklungsumgebung vertraut gemacht und nach dem Prinzip "soviel wie nötig" in die .NET-Philosophie eingeweiht. Nach der Lektüre dieses Kapitels und dem Nachvollziehen der abschließenden Praxisbeispiele sollte der Einsteiger über eine brauchbare Ausgangsbasis verfügen, um den sich vor ihm gewaltig auftürmenden Berg von Spezialkapiteln in Angriff zu nehmen.

Der erfahrene Visual Studio-Anwender erhält im Abschnitt 1.6 einen Überblick über die Neuerungen der Version 2012 gegenüber der Vorgängerversion Visual Studio 2010.

1.1 Die Installation von Visual Studio 2012

Ohne eine angemessen ausgestattete "Werkstatt" ist die Lektüre dieses Buchs nutzlos. Programmieren lernt man nur durch Beispiele, die man unmittelbar selbst am Rechner ausprobiert!

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buchs ist ein Rechner mit einer lauffähigen Installation von Visual Studio 2012!

1.1.1 Überblick über die Produktpalette

Alle im Handel angebotenen Visual-Studio-Pakete basieren auf dem .NET-Framework 4.5. Für welches der im Folgenden aufgeführten Produkte man sich entscheidet, hängt von den eigenen Anforderungen und Wünschen ab und ist nicht zuletzt auch eine Frage des Geldbeutels.

Visual Studio 2012 Express

Hier handelt es sich um abgespeckte, dafür aber kostenlose Versionen von Microsofts Entwicklungsumgebung. Wenn Sie als Hobby-Programmierer auf Features wie Berichte, Remote Debugging, ClickOnce etc. verzichten können, sind diese Minimalpakete in vielen Fällen ausreichend um eigene Anwendungen oder Webseiten zu erstellen. Folgende Editionen sind erhältlich:

- **Visual Studio Express 2012 für Windows 8**
  Mit dieser Edition können Sie WinRT-Applikationen für Windows 8 entwickeln. Enthalten sind neben C# auch Vorlagen für Visual Basic, JavaScript und C++, sowie das Windows 8 SDK und Blend für Visual Studio. Die von Ihnen geschriebenen Anwendungen können Sie anschließend im Windows Store Ihren potentiellen Kunden anbieten.
- **Visual Studio Express 2012 für Windows Desktop**
  Hiermit ist die einfache Entwicklung von Desktop- und Konsolenanwendungen für alle von Visual Studio 2012 unterstützten Windows-Versionen möglich. Neben C# stehen Ihnen dazu auch noch Visual Basic und C++ zur Verfügung. Enthalten sind auch einige Sprachtools (z.B. integrierter Unit-Test).
- **Visual Studio Express 2012 für Web**
  Bereits mit dieser Minimalausstattung ist die Entwicklung ansprechender interaktiver Webanwendungen möglich. Die Verteilung kann über den Webserver oder die Cloud unter Windows Azure erfolgen.
- **Visual Studio 2012 Express für Windows Phone**
  Unter Visual Studio 2012 wurde die Unterstützung für App-Entwickler nochmals stark erweitert. Eine entsprechende Visual Studio 2012 Express Edition wird allerdings erst mit der nächsten Version von Windows Phone verfügbar sein.
- **Visual Studio Team Foundation Server Express 2012**
  Teams mit bis zu fünf Entwicklern erhalten mit dieser Edition die Tools zur Quellcodeverwaltung, Buildautomatisierung undArbeitsaufgabennachverfolgung.

Visual Studio 2012 Professional

Wie es der Name bereits suggeriert, handelt es sich bei diesem Standard-Paket bereits um ein professionelles Werkzeug, denn es beinhaltet alle erforderlichen Kernfunktionen für die Entwicklung von Anwendungen für Windows, Office, das Web, die Cloud, Silverlight, SharePoint und Multi-Core-Szenarien.

Auch Visual Studio LightSwitch, die Entwicklungsumgebung für das Rapid Application Development (RAD), ist jetzt Bestandteil von Visual Studio Professional, Premium und Ultimate. Ähnliches gilt für die Team-Unterstützung und das Application Lifecycle Management (ALM), eine Sammlung von Tools und Prozessen zur Überwachung und Kontrolle des gesamten Entwicklungszyklus einer Applikation.

Sowohl ernstzunehmende Hobbyprogrammierer als auch professionelle Entwickler, die allein oder im kleinen Team an der Erstellung komplexer, mehrschichtiger Anwendungen arbeiten, sind mit dieser Edition gut beraten.

HINWEIS: Der Inhalt dieses Buches bezieht sich schwerpunktmäßig auf die Möglichkeiten der **Professional Edition**!

Visual Studio 2012 Premium

Bei diesem Paket handelt es sich um eine Vollausstattung für Softwareentwickler und -tester, um im Team Anwendungen auf Enterprise-Niveau zu entwickeln. Enthalten sind alle Funktionen der Professional-Version sowie weitere Funktionen, die komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen sollen. So finden sich

- Funktionen zur Verbesserung der Codequalität durch Codeüberprüfung mittels Peer-Workflow,
- bessere Entwicklungstools für den Entwurf von Multithreading-Anwendungen,
- Möglichkeiten zur Automatisieren von Benutzeroberflächentests
- Funktionen zum Suchen und Verwalten von doppeltem Code in der CodeBase zur Verbesserung der Architektur

Visual Studio 2012 Ultimate

Aufbauend auf dem Funktionsumfang von Visual Studio 2012 Premium finden sich zusätzlich folgende Funktionen:

- Zuverlässiges Erfassen und Reproduzieren von Fehlern, die während manueller und explorativer Tests gefunden werden, um nicht reproduzierbare Fehler zu vermeiden
- Verstehen der Abhängigkeiten und Beziehungen im Code durch Visualisierung
- Visualisieren der Auswirkung einer Änderung oder möglichen Änderung im Code
- Durchführen unbegrenzter Webleistungs- und Auslastungstests
- Entwerfen architektonischer Layerdiagramme zur Überprüfung des Codes und Implementierung in der Architektur

1.1.2 Anforderungen an Hard- und Software

Haben sich in der Vergangenheit die Hardwareanforderungen von Version zu Version in die Höhe geschraubt, so bleiben sie diesmal etwa auf dem gleichen Niveau wie beim Vorgänger Visual Studio 2010. Die folgende Auflistung kann lediglich eine Orientierungshilfe sein:

- Betriebssystem: Windows 8, Windows 7, Windows Server 2012, Windows Server 2008
- Unterstützte Architekturen: 32-Bit (x86) und 64-Bit (x64)
- Prozessor: 1,6-GHz-Pentium III+
- RAM: 1 GB verfügbarer physischer Arbeitsspeicher (x86) bzw. 2 GB (x64)
- Festplatte: 10 GB Speicherplatzbedarf
- Grafikkarte: DirectX 9-fähig mit einer Mindestauflösung von 1024 x 768 Pixeln
- DVD-ROM Laufwerk

Die Parameter von Prozessor und RAM sind als untere Grenzwerte zu verstehen, können aber für die Express-Editionen sicherlich noch etwas unterschritten werden.

Ganz wichtig:

HINWEIS: Wollen Sie WinRT-Anwendungen für Windows 8 entwickeln, so ist das Betriebssystem Windows 8 für das Entwicklungssystem unerlässlich!

Weiterhin ist zu beachten:

- Das neue .NET Framework 4.5 wird von Windows XP nicht mehr unterstützt – motten Sie also Ihren alten Computer ein.
- Das .NET-Framework 3.5 ist nicht mehr in Windows 8 enthalten, es muss nachinstalliert werden oder die Anwendungen müssen auf die Version 4 aktualisiert werden.
- Der SQL Server Express 2012 ist nicht mehr im Installationspaket enthalten, sondern muss separat heruntergeladen werden. Alternativ steht nach der Installation von Visual Studio der neue SQL Server Express 2012 LocalDB zur Verfügung

1.2 Unser allererstes C#-Programm

Jeder Weg, und ist er noch so weit, beginnt mit dem ersten Schritt! Nachdem die Mühen der Installation überstanden sind, wird es Zeit für ein allererstes C#-Programm. Wir verzichten allerdings auf das abgedroschene "Hello World" und wollen gleich mit etwas Nützlicherem beginnen, nämlich der Umrechnung von Euro in Dollar.

Auch allein mit dem .NET Framework SDK, also ohne das teure Visual Studio 2012, kann man vollwertige Programme entwickeln. Das wollen wir jetzt unter Beweis stellen, indem wir eine kleine Euro-Dollar-Applikation als so genannte *Konsolenanwendung* – dem einfachsten Anwendungstyp – schreiben.

1.2.1 Vorbereitungen

Voraussetzungen sind lediglich ein simpler Texteditor und der C#-Kommandozeilencompiler *csc.exe*.

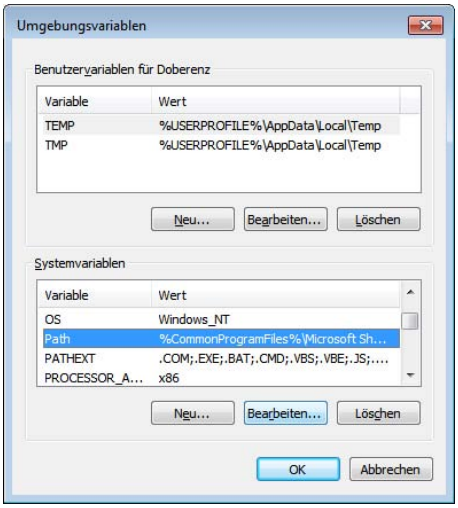
Compilerpfad eintragen

Der C#-Compiler *csc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

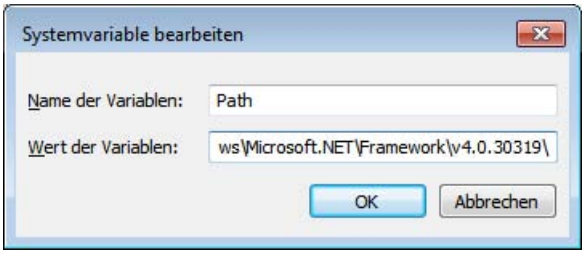
```
\\Windows\Microsoft.NET\Framework\v4.0.30319
```

Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *csc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- Sie finden den Dialog zur Einstellung der *Path*-Umgebungsvariablen in der Windows-Systemsteuerung unter dem Eintrag *System* im Aufgabenbereich "Erweiterte Systemeinstellungen".
- Im Dialog *Systemeigenschaften* klicken Sie auf der Registerkarte *Erweitert* auf die Schaltfläche *Umgebungsvariablen...*.



- Wählen Sie in der unteren Liste *Systemvariablen* den *Path*-Eintrag und klicken Sie auf die *Bearbeiten...*-Schaltfläche (siehe Abbildung).
- Hängen Sie den Namen des .NET Framework-Verzeichnisses, in welchem sich *csc.exe* befindet (*C:\Windows\Microsoft.NET\Framework\v4.0.30319*), durch ein Semikolon (;) getrennt hinten dran:

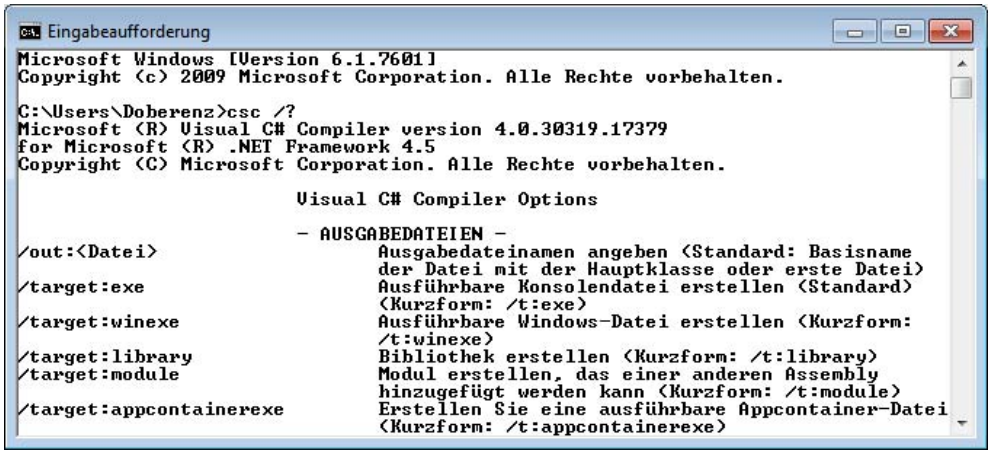


Die erfolgreiche Übernahme der Änderungen an den *Path*-Umgebungsvariablen können Sie in einem kleinen Test überprüfen, bei dem Sie sich als durchaus nützlichen Nebeneffekt gleich die vielfältigen Optionen des Compilers anzeigen lassen.

Wechseln Sie dazu über das Windows-Startmenü zur Eingabeaufforderung(*Start|Programme|Zubehör|Eingabeaufforderung*) und geben Sie (von einem beliebigen Verzeichnis aus) den folgenden Befehl ein, den Sie mit *Enter* abschließen:

```
csc /?
```

Aus der endlosen Parameterliste, die angezeigt wird, ist für uns die Option */target:exe* (abgekürzt */t:exe*) besonders interessant, da wir damit später unsere Konsolenanwendung kompilieren wollen (siehe Abbildung).



Vom Funktionieren des Compilers können Sie sich erst dann überzeugen, wenn Sie eine C#-Source-Datei erstellt haben (siehe folgender Abschnitt).

1.2.2 Quellcode schreiben

Öffnen Sie den im Windows-Zubehör enthaltenen Editor und tippen Sie, ohne lange darüber nachzudenken, einfach den folgenden Text ein:

```
using System;
class KonsolenDemo
{
  static void Main()
  {
    int i;
    Console.WriteLine("Umrechnung Euro in Dollar");
    do
    {
       float kurs, euro, dollar;
       Console.Write("Kurs 1 : ");
       kurs = Convert.ToSingle(Console.ReadLine());
       Console.Write("Euro: ");
       euro = Convert.ToSingle(Console.ReadLine());
       dollar = euro * kurs;
       Console.WriteLine("Sie erhalten " + dollar.ToString("0.00 $"));
       Console.Write("Programm beenden? (j/n)");
       string s = Console.ReadLine();
```

```
      i = string.Compare(s, "j");
    } while(i != 0);
  }
}
```

HINWEIS: Achten Sie auf die exakte Einhaltung der Groß-/Kleinschreibung!

Speichern Sie die Datei unter dem Namen *EuroDollar.cs* in ein extra dafür angelegtes Verzeichnis, z.B. *\EuroDollarKonsole*, ab.

1.2.3 Programm kompilieren und testen

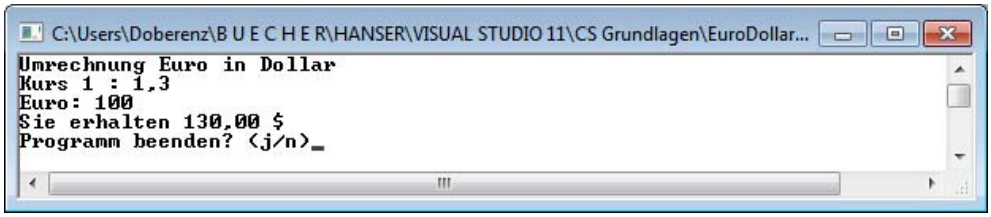
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus *...\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.cs* befindet.

Klicken Sie doppelt auf *cmd.exe* und rufen Sie dann den C#-Compiler wie folgt auf:

```
csc /t:exe EuroDollar.cs
```

Nach dem erfolgreichen Kompilieren wird sich eine neue Datei *EuroDollar.exe* im Anwendungsverzeichnis befinden, ansonsten gibt der Compiler eine Fehlermeldung aus.

Klicken Sie doppelt auf die Datei *EuroDollar.exe* und führen Sie Ihr erstes C#-Programm aus!



1.2.4 Einige Erläuterungen zum Quellcode

Da wir auf die Grundlagen der Sprache C# erst in den späteren Kapiteln ausführlich zu sprechen kommen, sollen die folgenden Informationen zunächst nur allererste Eindrücke vermitteln.

Befehlszeilen und Gültigkeitsbereiche

Das Ende einer C#-Befehlszeile wird durch ein Semikolon (;) markiert. Der Zeilenumbruch spielt also keine Rolle! Gültigkeitsbereiche sind durch geschweifte Klammern { ... } eingegrenzt. In der Regel muss also die Anzahl der öffnenden Klammern gleich der Anzahl schließender Klammern sein.

using-Anweisung

Mit der ersten Anweisung

```
using System;
```

binden Sie den *System*-Namensraum (*Namespace*) ein. Das hat den Vorteil, dass Sie statt

```
System.Console.WriteLine("Umrechnung Euro in Dollar");
```

nur noch

```
Console.WriteLine("Umrechnung Euro in Dollar");
```

schreiben müssen.

class-Anweisung

Mit dieser Anweisung erzeugen Sie eine neue Klasse, in welcher in unserem Beispiel die *Main*-Prozedur deklariert wird. Diese definiert den Einsprungpunkt der Konsolenanwendung (also dort, wo das Programm startet).

HINWEIS: Ein C#-Programm besteht aus mindestens einer **.cs*-Textdatei mit einer Klasse.

WriteLine- und ReadLine-Methoden

Diese Methoden der *Console*-Klasse erlauben die Aus- und Eingabe von Text.

Während *Write* nur den Text an der aktuellen Position ausgibt, erzeugt *WriteLine* zusätzlich einen Zeilenumbruch.

ReadLine erwartet die Betätigung der *Enter*-Taste und liefert die vorher eingegebenen Zeichen als Zeichenkette zurück.

Assemblierung

Bei der vom C#-Compiler erzeugten Datei *EuroDollar.exe* handelt es sich **nicht** um eine herkömmliche Exe-Datei, sondern um eine so genannte *Assemblierung*, die erst in Zusammenarbeit mit der CLR (*Common Language Runtime*) des .NET-Frameworks in Maschinencode verwandelt wird (siehe Abschnitt 1.5.2).

1.2.5 Konsolenanwendungen sind langweilig

Zwar hat seit Visual C# 2005 die Klasse *System.Console* zahlreiche neue Mitglieder erhalten, mit denen auch verschiedenste farbliche Effekte möglich sind, trotzdem: Bei wem weckt das Outfit einer Konsolenanwendung noch positive Emotionen[1]?

---

[1] Von nostalgischen Erinnerungen an die DOS-Steinzeit einmal abgesehen.

Als einfaches Hilfsmittel zum Erlernen von C# und als Notnagel für all jene, die sich Visual Studio 2012 vorerst nicht leisten wollen oder können, hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2012 werden wir im Praxisteil dieses Kapitels (Abschnitt 1.7.2) das gleiche Problem nochmals lösen, dann allerdings mit einer attraktiven Windows-Oberfläche. Bevor es aber damit losgehen kann, sollten Sie sich ein wenig mit der Windows-Philosophie anfreunden.

1.3 Die Windows-Philosophie

Eine moderne Programmiersprache wie C# gibt Ihnen die faszinierende Möglichkeit, eigene Windows-Programme mit relativ geringem Aufwand und nach nur kurzer Einarbeitungszeit selbst zu entwickeln. Allerdings fällt der Einstieg umso leichter, je schneller man sich Klarheit über die zunächst fremdartig anmutende, aber dann doch einfach und gleichzeitig genial erscheinende Windows-Philosophie verschafft.

1.3.1 Mensch-Rechner-Dialog

Die Art und Weise, **wie** die Kommunikation mit dem Benutzer (Mensch-Rechner-Dialog) abläuft, dürfte wohl der gravierendste Unterschied zwischen einer klassischen Konsolenanwendung und einer typischen Windows-Anwendung sein. Wie Sie es bereits im Einführungsbeispiel 1.2 kennen gelernt haben, "wartet" das Konsolenprogramm auf eine Eingabe, indem die Tastatur zyklisch abgefragt wird.

Unter Windows werden hingegen Ein- und Ausgaben in so genannte "Nachrichten" umgesetzt, die zum Programm geschickt und dort in einer Nachrichtenschleife kontinuierlich verarbeitet werden. Daraus ergibt sich ein grundsätzlich anderes Prinzip der Interaktion zwischen Mensch und Rechner:

- Während bei einer Konsolenanwendung alle Initiativen für die Benutzerkommunikation vom Programm ausgehen, hat in einer Windows-Anwendung der Bediener den Hut auf. Er bestimmt durch seine Eingaben den Ablauf der Rechnersitzung.
- Während eine Konsolenanwendung in der Regel in einem einzigen Fenster läuft, erfolgt die Ausgabe bei einer Windows-Anwendung meist in mehreren Fenstern.

1.3.2 Objekt- und ereignisorientierte Programmierung

Vergleicht man den Programmaufbau einer Konsolenanwendung, welche aus einer langen Liste von Anweisungen besteht, mit einer Windows-Anwendung, so stellt man folgende Hauptunterschiede fest:

- Im Konsolenprogramm werden die Befehle sequenziell abgearbeitet, d h. Schritt für Schritt hintereinander. Den Gesamtablauf kontrolliert in der Regel ein Hauptprogramm.

- In einer Windows-Anwendung laufen alle Aktionen objekt- und ereignisorientiert ab, eine streng vorgeschriebene Reihenfolge für die Eingabe und Abarbeitung der Befehle gibt es nicht mehr. Für jede Aktivität des Anwenders ist ein Programmteil zuständig, der weitestgehend unabhängig von anderen Programmteilen agieren kann und muss. Daraus folgt auch das Fehlen eines Hauptprogramms im herkömmlichen Sinn!

Ein Windows-Programmierer hat sich vor allem mit den folgenden Begriffen auseinander zu setzen:

Objekte (Objects)

Das sind zunächst die Elemente der Windows-Bedienoberfläche, denen wiederum Eigenschaften, Ereignisse und Methoden zugeordnet werden.

Beschränken wir uns der Einfachheit halber zunächst nur auf die visuelle Benutzerschnittstelle, so haben wir es in C# mit folgenden Objekten zu tun:

- **Formulare:**
  Das sind die Fenster, in welchen eine C#-Anwendung ausgeführt wird. In einem Formular (*Form*) können weitere untergeordnete Formulare, Komponenten (siehe unten), Text oder Grafik enthalten sein.
- **Steuerelemente:**
  Diese tauchen in vielfältiger Weise als Schaltflächen (*Button*), Textfelder (*TextBox)* etc. auf. Sie stellen die eigentliche Benutzerschnittstelle dar, über welche mittels Tastatur oder Maus Eingaben erfolgen oder die der Ausgabe von Informationen dienen.

Der Objektbegriff wird auch auf die nichtvisuellen Elemente (z.B *Timer, DataSet...*) ausgedehnt, und das geht schließlich so weit, dass innerhalb des .NET-Frameworks sogar alle Variablen als Objekte betrachtet werden. Natürlich dürfen auch Sie als Programmierer auch eigene Objekte/ Komponenten entwickeln und hinzufügen.

Eigenschaften (Properties)

Unter diesem Begriff versteht man die Attribute von Objekten, wie z.B. die Höhe (*Height*) und die Breite (*Width*) oder die Hintergrundfarbe (*BackColor*) eines Formulars. Jedes Objekt verfügt über seinen eigenen Satz von Eigenschaften, die teilweise nur zur Entwurfs- oder nur zur Laufzeit veränderbar sind.

Methoden (Methods)

Das sind die im Objekt definierten Funktionen und Prozeduren, die gewissermaßen das "Verhalten" beim Eintreffen einer Nachricht bestimmen. So säubert z.B. die *Clear*-Methode den Inhalt einer *ListBox*. Eine Methode kann z.B. auch das Verhalten des Objekts bei einem Mausklick, einer Tastatureingabe oder sonstigen Ereignissen (siehe unten) definieren. Im Unterschied zu den oben genannten Eigenschaften (Properties), die eine "statische" Beschreibung liefern, bestimmen Methoden die "dynamischen" Fähigkeiten des Objekts.

Ereignisse (Events)

Dies sind Nachrichten, die vom Objekt empfangen werden. Sie stellen die eigentliche Windows-Schnittstelle dar. So ruft z.B. das Anklicken eines Steuerelements mit der Maus in Windows ein *Click*-Ereignis hervor. Aufgabe eines Windows-Programms ist es, auf alle Ereignisse gemäß dem Wunsch des Anwenders zu reagieren. Dies geschieht in so genannten *Ereignisbehandlungsroutinen* (Eventhandler).

Diese (zugegebenermaßen ziemlich oberflächlichen und unvollständigen) Erklärungen zur objektorientierten Programmierung sollen vorerst zum Einstieg genügen, theoretisch sauber wird die OOP erst im Kapitel 3 erläutert.

1.3.3 Programmieren mit Visual Studio 2012

Nicht nur Konsolenanwendungen, sondern auch Windows- und Web-Anwendungen lassen sich rein theoretisch mit den (kostenlos erhältlichen) Werkzeugen des *Microsoft .NET Framework SDK* erstellen. Allerdings ist dies extrem umständlich, da dazu zeitaufwändige Überlegungen zur Gestaltung der Benutzerschnittstelle[1] und ständiges Nachschlagen in der Dokumentation erforderlich wären. Die intuitive Entwicklungsumgebung Visual Studio befreit Sie von diesem, besonders bei größeren Projekten sehr lästigen und nervtötenden Herumwursteln und erlaubt (unabhängig von der verwendeten Programmiersprache) eine systematische Vorgehensweise in vier Etappen:

1. Visueller Entwurf der Bedienoberfläche
2. Zuweisen der Objekteigenschaften
3. Verknüpfen der Objekte mit Ereignissen
4. Kompilieren und Testen der Anwendung

Bereits die *erste Etappe* weist einen deutlichen Unterschied zur Konsolenprogrammiertechnik auf: Am Anfang steht der Oberflächenentwurf!

Ausgangsbasis ist das vom Editor bereitgestellte Startformular (*Form1*), welches mit diversen Steuerelementen, wie Schaltflächen (*Buttons*) oder Editierfenstern (*TextBox*en), ausgestattet wird. Im Werkzeugkasten finden Sie ein nahezu komplettes Angebot der Windows-typischen Steuerelementen. Diese werden ausgewählt, mittels Maus an ihre endgültige Position gezogen und (falls nötig) in ihrer Größe verändert.

Bereits während der ersten Etappe hat man, mehr oder weniger unbewusst, Eigenschaften, wie Position und Abmessungen von Formularen und Steuerelementen, verändert. In der *zweiten Etappe* braucht man sich eigentlich nur noch um die Eigenschaften zu kümmern, die von den Standardeinstellungen (Defaults) abweichen.

Die *dritte Etappe* haucht Leben in unsere bislang nur mit statischen Attributen ausgestatteten Objekte. Hier muss in so genannten *Ereignisbehandlungsroutinen* (Eventhandlern) festgelegt werden, **wie** das Formular oder das betreffende Steuerelement auf bestimmte Ereignisse zu reagieren hat. Visual Studio stellt auch hier "vorgefertigten" Rahmencode (erste und letzte Anweisung)

[1] Das geht hin bis zum Abzählen von Pixeln!

für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit C#-Quellcode aus. Hier können Methoden oder Prozeduren aufgerufen werden, aber auch Eigenschaften anderer Objekte lassen sich während der Laufzeit neu zuweisen.

In der *vierten Etappe* schlägt schließlich die Stunde der Wahrheit. Das von Ihnen geschriebene Programm wird vom C#-Compiler in einen Zwischencode übersetzt und läuft damit auf jedem Rechner, auf dem das .NET Framework installiert ist.

Allerdings ist die Arbeit des Programmierers nur in seltenen Fällen bereits nach einmaligem Durchlaufen aller vier Etappen getan. In der Regel müssen Fehler ausgemerzt und Ergänzungen vorgenommen werden, sodass sich der beschriebene Entwicklungszyklus auf ständig höherem Level so lange wiederholt, bis ein zufrieden stellendes Ergebnis erreicht ist.

Der in diesem Zyklus praktizierte visuelle Oberflächenentwurf, verbunden mit dem ereignisorientierten Entwurfskonzept, macht *Visual Studio 2012* zu einer hocheffektiven Entwicklungsumgebung für Windows- und Web-Anwendungen.

1.4 Die Entwicklungsumgebung Visual Studio 2012

Visual Studio 2012 ist eine universelle Entwicklungsumgebung (IDE[1]) für Windows- und für Web-Anwendungen, die auf Microsofts .NET-Technologie basieren. Alle notwendigen Tools, wie z.B. für den visuellen Oberflächenentwurf, für die Codeprogrammierung und für die Fehlersuche, werden bereitgestellt.

C# ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2012 einsetzen können. So werden z.B. noch Visual Basic, Visual C++ und Visual F# unterstützt.

HINWEIS: Der vorliegende Abschnitt soll lediglich einen allerersten Eindruck der IDE vermitteln, der sich erst durch die konkrete Arbeit mit den Praxisbeispielen am Ende dieses Kapitels verfestigen wird!

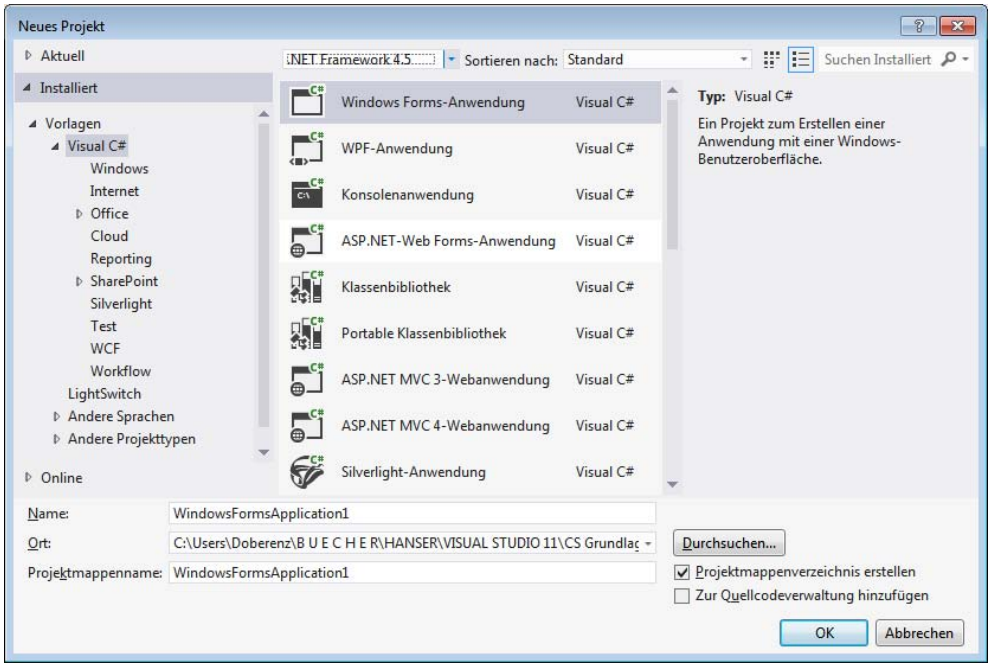
1.4.1 Neues Projekt

Wählen Sie auf der Startseite von Visual Studio 2012 den Menüpunkt *Neues Projekt...,* so öffnet sich der Startdialog *Neues Projekt* mit einem umfangreichen und zunächst verwirrenden Angebot an unterschiedlichen Vorlagen[2] für Visual C#-Projekttypen, wobei für den Einsteiger zunächst die klassische *Windows Forms-Anwendung* empfohlen wird.

Visual Studio 2012 erlaubt es, Programme für verschiedene .NET-Framework-Versionen zu entwickeln (Multi-Targeting, siehe obere Klappbox).

[1] *Integrated Developers Environment*

[2] Die Abbildung bezieht sich auf die Professional-Edition, bei den anderen Editionen von Visual Studio 2012 ist das Angebot an unterschiedlichen Projekttypen bzw. Vorlagen mehr oder weniger eingeschränkt.



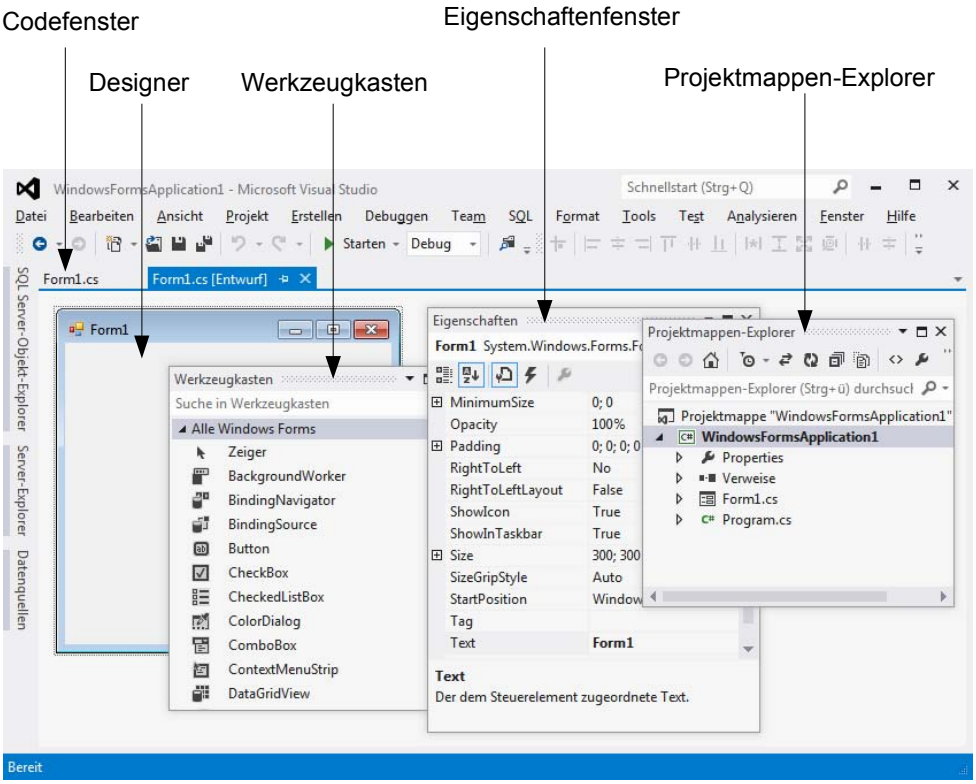
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

HINWEIS: Namen und Ort des Projekts sollten Sie unbedingt **vor** dem Klicken der *OK*-Schaltfläche eintragen, denn ein späteres Umbenennen ist recht umständlich.

1.4.2 Die wichtigsten Fenster

Haben Sie als Projekttyp beispielsweise *Windows Forms-Anwendung* gewählt, könnte Ihnen Visual Studio 2012 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.

HINWEIS: Falls sich eines der Fenster versteckt hat, können Sie es jederzeit über das *Ansicht*-Menü herbeiholen.



Der Projektmappen-Explorer

Da es unter Visual Studio 2012 möglich ist, mehrere Projekte gleichzeitig zu bearbeiten, gibt es eine Projektmappendatei mit der Extension *.sln* (Solution), deren Inhalt im Projektmappen-Explorer (*STRG+R*) übersichtlich angezeigt wird. Sie können dieses Fenster deshalb ohne Übertreibung als "Schaltzentrale" Ihres Projekts betrachten.

Die zu jedem einzelnen C#-Projekt gehörigen Dateien und Einstellungen werden in einer XML-Datei mit der Extension *.csproj* (C#-Projekt) verwaltet.

HINWEIS: Öffnen Sie Ihre Projekte immer über die *.sln*-Projektmappendatei, statt über die *.csproj*-Projektdatei, selbst wenn nur ein einziges Projekt enthalten ist!

Zur Bedeutung der einzelnen Einträge bzw. Dateien:

- *Properties*
  Hier sind verschiedene Dateien zusammengefasst, die die Projekteigenschaften bestimmen. *AssemblyInfo.cs* enthält z.B allgemeine Infos zur Assemblierung des Projekts, wie Titel, Beschreibung, Versionsnummer, Copyright. Weitere Dateien beziehen sich auf die Ressourcen und die Projekteinstellungen.

- *Verweise*
  Hier sind die aktuell für das Projekt gültigen Verweise auf Namensräume bzw. Assemblierungen enthalten. Standardmäßig hat Visual Studio bereits die wichtigsten Verweise eingestellt, weitere können Sie über das Kontextmenü der rechten Maustaste hinzufügen.
- *Form1.cs*
  Diese Datei enthält eine partielle Klasse[1], die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.
- *Form1.Designer.cs*
  Diese Datei enthält eine partielle Klasse, die den vom Windows Forms Designer automatisch generierten Code von *Form1* kapselt. Der gesamte Code von *Form1* ist also zwischen den partiellen Klassen in *Form1.cs* und *Form1.Designer.cs* aufgeteilt.
- *Program.cs*
  Eine statische Klasse, welche die *Main*-Methode (den Einsprungpunkt der Anwendung) enthält. In dieser Methode wird durch Aufruf von *Application.Run* eine Nachrichtenschleife gestartet, die ununterbrochen auf Ereignisse wartet, damit die Anwendung darauf reagieren kann.

Durch Doppelklick auf eine dieser Dateien können Sie diese im Designer bzw. im Codefenster zwecks Bearbeitung öffnen.

Der Designer

Im Designer-Fenster entwerfen Sie die Programmoberfläche bzw. Benutzerschnittstelle. Ähnlich wie bei einem Zeichenprogramm entnehmen Sie dem Werkzeugkasten Steuerelemente und ziehen diese per Drag & Drop auf ein Formular. Hier können Sie weitere Eigenschaften, wie z.B. Größe und Position, direkt mit der Maus und andere, wie z.B. Farbe und Schriftart, über das Eigenschaften-Fenster ändern.

Der Werkzeugkasten

Der Werkzeugkasten wird häufig benötigt (Menü *Ansicht/Werkzeugkasten* bzw. *STRG+W, X*). Auf verschiedenen Registerseiten, die später von Ihnen auch frei konfiguriert werden können, finden Sie eine umfangreiche Palette verschiedenster Steuerelemente für Windows-Forms-Anwendungen.

Das Eigenschaften-Fenster

Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. *F4*) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[2]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können.

Das Aktivieren eines bestimmten Steuerelements geschieht entweder durch Anklicken desselben auf dem Formular, oder durch dessen Auswahl in der Klappbox am oberen Rand des Eigenschaften-Fensters.

---

[1] Das Konzept partieller Klassen wird im OOP-Kapitel (Abschnitt 3.7 3) erläutert.

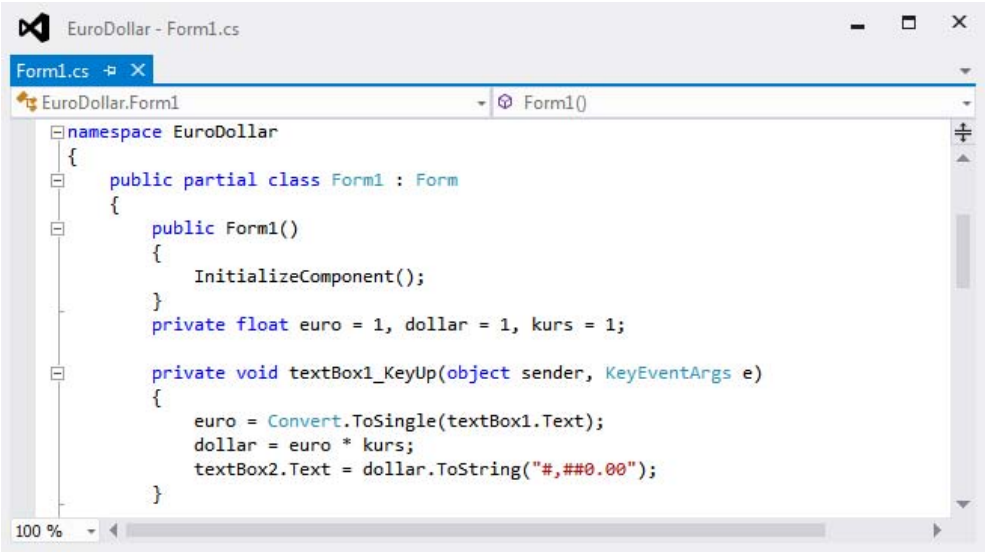
[2] Lassen Sie sich nicht davon irritieren, dass das Eigenschaftenfenster auf einer extra Registerseite auch die zum Steuerelement gehörigen Ereignisse anbietet.

Das Codefenster

Für die eigentliche Programmierung ist das Codefenster zuständig. Logischerweise wird dies damit auch zu Ihrem Hauptbetätigungsfeld als C#-Programmierer. Um zum Beispiel das zu einem Formular gehörige Codefenster zu öffnen, klicken Sie auf dasselbe mit der rechten Maustaste und wählen im Kontextmenü *Code anzeigen* (F7). Die folgende Abbildung zeigt das Codefenster für *Form1* im Praxisbeispiel 1.7.2.

Der Code-Editor unterstützt Sie auf vielfältige Weise beim Schreiben von Quellcode, so markiert er Wörter farblich, unterbreitet Ihnen Vorschläge, weist Sie auf Fehler hin oder rückt den Text automatisch ein.

Bei allem Verständnis für Ihre Ungeduld: bevor Sie mit praktischen Beispielen beginnen, empfehlen wir Ihnen zunächst eine kleine Exkursion in die Untiefen von .NET.



1.5 Microsofts .NET-Technologie

Ganz ohne Theorie geht nichts! In diesem leider etwas "trockenen" Abschnitt wollen wir den Einsteiger mit der grundlegenden .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features vertraut machen. Dazu dürfen Sie Ihrem Rechner ruhig einmal eine Pause gönnen.

1.5.1 Zur Geschichte von .NET

Das Kürzel .NET ist die Bezeichnung für eine gemeinsame Plattform für viele Programmiersprachen. Beim Kompilieren von .NET-Programmen wird der jeweilige Quelltext in MSIL (*Microsoft Intermediate Language*) übersetzt. Es gibt nur ein gemeinsames Laufzeitsystem für alle Sprachen, die so genannte CLR (*Common Language Runtime*), das die MSIL-Programme ausführt.

Die im Jahr 2002 eingeführte .NET-Technologie wurde deshalb notwendig, weil sich die Anforderungen an moderne Softwareentwicklung dramatisch verändert hatten, wobei das rasant wachsende Internet mit seinen hohen Ansprüchen an die Skalierbarkeit einer Anwendung, die Verteilbarkeit auf mehrere Schichten und ausreichende Sicherheit der hauptsächliche Motor war, sich nach einer grundlegend neuen Sprachkonzeption umzuschauen.

Mit .NET fand ein radikaler Umbruch in der Geschichte der Softwareentwicklung statt. Nicht nur dass jetzt "echte" objektorientierte Programmierung zum obersten Dogma erhoben wird, nein, auch eine langjährig bewährte Sprache wie das alte Visual Basic wurde völlig umgekrempelt und die einst hoch gelobte COM-Technologie zum Auslaufmodell erklärt!

Warum eine extra Programmiersprache?

C# wurde ausschließlich für das .NET-Framework konzipiert, wobei versucht wurde, das Beste aus den etablierten Programmiersprachen Java, JavaScript, Visual Basic und C++ zu kombinieren, ohne aber deren Nachteile zu übernehmen.

Da sich die etablierten Sprachen nicht ohne größere Kompromisse an das .NET-Framework anpassen ließen, haben die .NET-Entwickler die Gelegenheit beim Schopf gepackt und eine "maßgeschneiderte" .NET-Sprache entwickelt. C# setzt sauber auf dem .NET-Framework auf und kommt ohne "faule" Kompatibilitäts-Kompromisse aus, wie sie z.B. teilweise bei Visual Basic erforderlich waren.

Ein großer Teil der .NET-Klassenbibliotheken ist selbst in C# programmiert, C# ist somit die Systemsprache von .NET und spielt gewissermaßen die gleiche Rolle wie C++ für Windows.

Als konsequent objektorientierte Sprache erfüllt C# folgende Kriterien:

- **Abstraktion**
  Die Komplexität eines Geschäftsproblems ist beherrschbar, indem eine Menge von abstrakten Objekten identifiziert werden können, die mit dem Problem verknüpft sind.
- **Kapselung**
  Die interne Implementation einer solchen Abstraktion wird innerhalb des Objekts versteckt.
- **Polymorphie**
  Ein und dieselbe Methode kann auf mehrere Arten implementiert werden.
- **Vererbung**
  Es wird nicht nur die Schnittstelle, sondern auch der Code einer Klasse vererbt (Implementations-Vererbung statt der COM-basierten Schnittstellen-Vererbung).

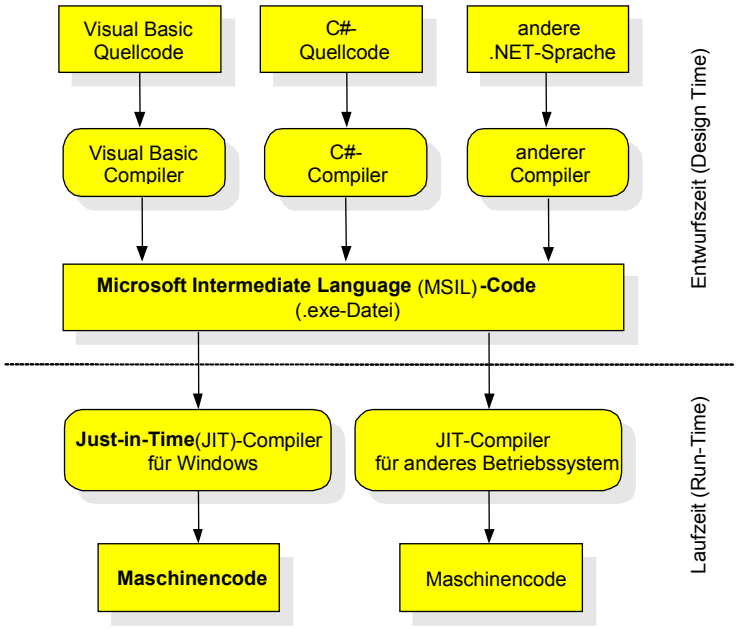
Microsoft kann natürlich nicht über Nacht die COM-Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer würden dadurch auf immer und ewig verprellt werden und sich frustriert einer stabileren Entwicklungsplattform zuwenden. Aus diesem Grund wird COM auch in .NET noch einige Zeit sein Gnadenbrot erhalten.

Wie funktioniert eine .NET-Sprache?

Jeder in einer beliebigen .NET-Programmiersprache geschriebene Code wird beim Kompilieren in einen Zwischencode, den so genannten MSIL-Code (*Microsoft Intermediate Language Code*), übersetzt, der unabhängig von der Plattform bzw. der verwendeten Hardware ist und dem man es auch nicht mehr ansieht, in welcher Sprache seine Source geschrieben wurde.

HINWEIS: Das .NET-Konzept sieht fast wie ein Java-Plagiat aus, allerdings mit dem "feinen" Unterschied, dass es nicht an eine bestimmte Programmiersprache gebunden ist!

Erst wenn der MSIL-Code von einem Programm zur Ausführung genutzt werden soll, wird er vom *Just-in-Time(JIT)-Compiler* in Maschinencode übersetzt[1]. Ein .NET-Programm wird also vom Entwurf bis zu seiner Ausführung auf dem Zielrechner tatsächlich zweimal kompiliert (siehe folgende Abbildung).



HINWEIS: Für die Installation eines Programms ist in der Regel lediglich die Weitergabe des MSIL-Codes erforderlich. Voraussetzung ist allerdings das Vorhandensein der .NET-Laufzeitumgebung (CLR), die Teil des .NET Frameworks ist, auf dem Zielrechner.

[1] Der Begriff "jeder Code" schließt z.B. auch den Code der ASP.NET-Seiten ein.

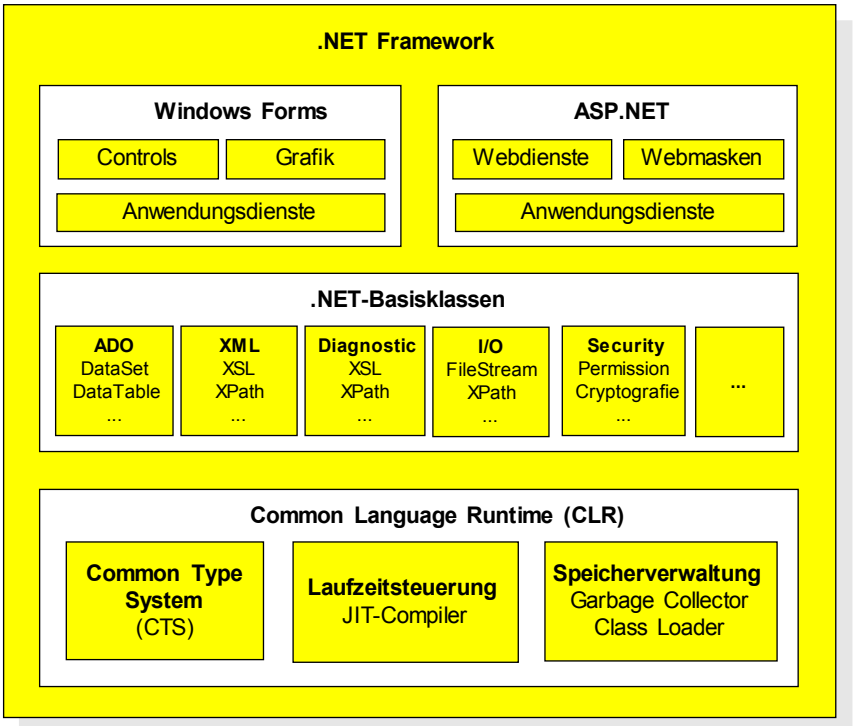
1.5.2 .NET-Features und Begriffe

Mit Einführung von Microsofts .NET-Technologie prasselte auch eine Vielzahl neuer Begriffe auf die Entwicklergemeinde ein. Wir wollen hier nur die wichtigsten erklären.

.NET-Framework

.NET ist die Infrastruktur für die gesamte .NET-Plattform, es handelt sich hierbei gleichermaßen um eine Entwurfs- wie um eine Laufzeitumgebung, in welcher Windows- und Web-Anwendungen erstellt und verteilt werden können.

Die nachfolgende Abbildung versucht, einen groben Überblick über die Komponenten des .NET Frameworks zu geben.



Zu den wichtigsten Komponenten des .NET-Frameworks und den damit zusammenhängenden Begriffen zählen:

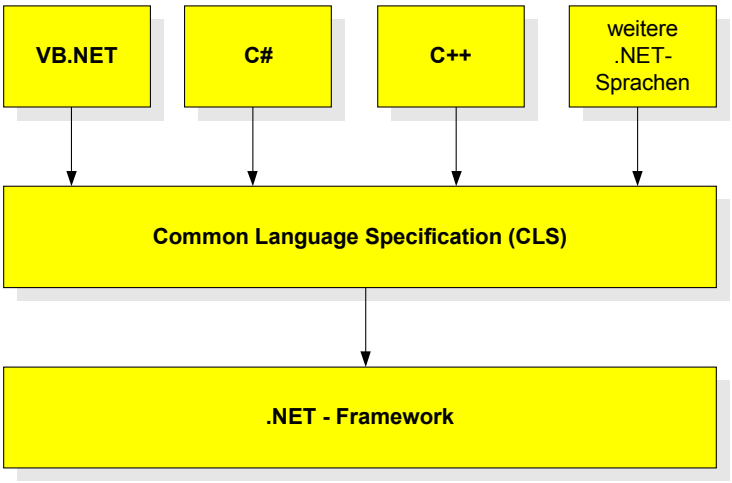
- Common Language Specification (CLS)
- Common Type System (CTS)
- Common Language Runtime (CLR)
- .NET-Klassenbibliothek

- diverse Basisklassenbibliotheken wie ADO.NET und ASP.NET
- diverse Compiler z.B. für C#, VB.NET ...

Im Folgenden sollen die einzelnen .NET-Bestandteile einer näheren Betrachtung unterzogen werden.

Die Common Language Specification (CLS)

Um den sprachunabhängigen MSIL-Zwischencode erzeugen zu können, müssen allgemein gültige Richtlinien und Standards für die .NET-Programmiersprachen existieren. Diese werden durch die *Common Language Specification* (CLS) definiert, die eine Reihe von Eigenschaften festlegt, die jede .NET-Programmiersprache erfüllen muss.



HINWEIS: Ganz egal, mit welcher .NET-Programmiersprache Sie arbeiten, der Quellcode wird immer in ein und dieselbe Intermediate Language (MSIL) kompiliert.

Besonders für die Entwicklung von .NET-Anwendungen im Team haben die Standards der CLS weitreichende positive Konsequenzen, denn es ist nun zweitrangig, in welcher .NET-Programmiersprache Herr Müller die Komponente X und Herr Meier die Komponente Y schreibt. Alle Komponenten werden problemlos miteinander interagieren!

Auf einen wichtigen Bestandteil des CLS kommen wir im folgenden Abschnitt zu sprechen.

Das Common Type System (CTS)

Ein Kernbestandteil der CLS ist das *Common Type System (CTS)*, es definiert alle Typen[1], die von der .NET-Laufzeitumgebung (CLR) unterstützt werden.

[1] Unter .NET spricht man allgemein von Typen und meint damit Klassen, Interfaces und Datentypen, die als Wert übergeben werden.

Alle diese Typen lassen sich in zwei Kategorien aufteilen:

- Wertetypen (werden auf dem Stack abgelegt)
- Referenztypen (werden auf dem Heap abgelegt)

Zu den Wertetypen gehören beispielsweise die ganzzahligen Datentypen und die Gleitkommazahlen, zu den Referenztypen zählen die Objekte, die aus Klassen instanziiert wurden.

HINWEIS: Dass unter .NET auch die Wertetypen letztendlich als Objekte betrachtet und behandelt werden, liegt an einem als *Boxing* bezeichneten Verfahren, das die Umwandlung eines Werte- in einen Referenztypen zur Laufzeit besorgt.

Warum hat Microsoft mit dem heiligen Prinzip der Abwärtskompatibilität gebrochen und selbst die fundamentalen Datentypen einer Programmiersprache neu definiert?

Als Antwort kommen wir noch einmal auf eine wesentliche Säule der .NET-Philosophie zu sprechen, auf die durch CLS/CTS manifestierte Sprachunabhängigkeit und auf die Konsequenzen, die dieses neue Paradigma nach sich zieht.

Microsofts .NET-Entwickler hatten gar keine andere Wahl, denn um Probleme beim Zugriff auf sprachfremde Komponenten zu vermeiden und um eine sprachübergreifende Programmentwicklung überhaupt zu ermöglichen, mussten die Spezifikationen der Programmiersprachen durch die CLS einander angepasst werden. Dazu müssen alle wesentlichen sprachbeschreibenden Elemente – wie vor allem die Datentypen – in allen .NET-Programmiersprachen gleich sein.

Da .NET eine Normierung der Programmiersprachen erzwingt, verwischen die Grenzen zwischen den verschiedenen Sprachen, und Sie brauchen nicht immer umzudenken, wenn Sie tatsächlich einmal auf eine andere .NET-Programmiersprache umsteigen wollen.

Als Lohn für die Mühen und den Mut, die eingefahrenen Gleise seiner altvertrauten Sprache zu verlassen, winken dem Entwickler wesentliche Vereinfachungen. So sind die Zeiten des alltäglichen Ärgers mit den Datentypen – wie z.B. bei der Übergabe eines Integers an eine C-DLL – endgültig vorbei. Bei so viel Licht gibt es natürlich auch Schatten:

HINWEIS: Mit einem der inzwischen zahlreich und kostenlos verfügbaren MSIL-Decompiler ist es sehr einfach möglich, aus einer EXE-Datei den Quellcode zu generieren. Wenn überhaupt, so ist es nur mit hohem Aufwand möglich, dass Sie als Programmierer Ihr Know-how vor der Konkurrenz schützen können.

Die Common Language Runtime (CLR)

Die Laufzeitumgebung bzw. *Common Language Runtime* (CLR) ist die Umgebung, in welcher .NET-Programme auf dem Zielrechner ausgeführt werden, sie muss auf einem Computer nur einmal installiert sein, und schon laufen sämtliche .NET-Anwendungen, egal ob sie in C#, VB.NET oder F# programmiert wurden. Die CLR zeichnet für die Ausführung der Anwendungen verantwortlich und kooperiert auf Basis des CTS mit der MSIL.

Mit ihren Fähigkeiten bildet die *Common Language Runtime* (CLR) gewissermaßen den Kern von .NET. Den Code, der von der CLR ausgeführt wird, bezeichnet man auch als verwalteten bzw. *Managed Code*.

Die CLR ist innerhalb des .NET-Frameworks nicht nur für das Ausführen von verwaltetem Code zuständig, der Aufgabenbereich der CLR ist weitaus umfangreicher und umfasst zahlreiche Dienste, die als Bindeglied zwischen dem verwalteten MSIL-Code und dem Betriebssystem des Rechners die Anforderungen des .NET-Frameworks sicherstellen, wie z.B.

- ClassLoader
- Just-in-Time(JIT)-Compiler
- ExceptionManager
- Code Manager
- Security Engine
- Debug Machine
- Thread Service
- COM-Marshaller

Die Verwendung der sprachneutralen MSIL erlaubt die Nutzung des CTS und der Basisklassen für alle .NET-Sprachen gleichermaßen. Einziger hardwareabhängiger Bestandteil des .NET-Frameworks ist der Just-in-Time Compiler. Deshalb kann der MSIL-Code im Prinzip frei zwischen allen Plattformen bzw. Geräten, für die ein .NET Framework existiert, ausgetauscht werden.

Namespaces ersetzen Registry

Alle Typen des .NET-Frameworks werden in so genannten Namensräumen (Namespaces) zusammengefasst. Unabhängig von irgendeiner Klassenhierarchie wird jede Klasse einem bestimmten Anwendungsgebiet zugeordnet.

Die folgende Tabelle zeigt beispielhaft einige wichtige Namespaces für die Basisklassen des .NET-Frameworks:

Namespace	... enthält Klassen für ...
System.Windows.Forms	... Windows-basierte Anwendungen
System.Collections	... Objekt-Arrays
System.Drawing	... die Grafikprogrammierung
System.Data	... den ADO-Datenbankzugriff
System.Web	... die HTTP-Webprogrammierung
System.IO	... Ein- und Ausgabeoperationen

Mit den Namespaces hat auch der Ärger mit der Registrierung von (COM-)Komponenten bei Versionskonflikten sein Ende gefunden, denn eine unter .NET geschriebene Komponente wird von

der .NET-Runtime nicht mehr über die ProgID der Klasse mit Hilfe der Registry lokalisiert, sondern über einen in der Runtime enthaltenen Mechanismus, welcher einen Namespace einer angeforderten Komponente sowie deren Version für das Heranziehen der "richtigen" Komponente verwendet.

Assemblierungen

Unter einer Assemblierung (*Assembly*) versteht man eine Basiseinheit für die Verwaltung von Managed Code und für das Verteilen von Anwendungen, sie kann sowohl aus einer einzelnen als auch aus mehreren Dateien (Modulen) bestehen. Eine solche Datei (*.dll* oder *.exe*) enthält MSIL-Code (kompilierter Zwischencode).

Die Klassenverwaltung in Form von selbst beschreibenden Assemblies vermeidet Versionskonflikte von Komponenten und ermöglicht vor allem dynamische Programminstallationen aus dem Internet. Statt der bei einer klassischen Installation bisher erforderlichen Einträge in die Windows-Registry genügt nunmehr einfaches Kopieren der Anwendung.

Normalerweise müssen Sie die Assemblierungen referenzieren, in welchen die von Ihrem Programm verwendeten Typen bzw. Klassen enthalten sind. Eine Ausnahme ist die Assemblierung *mscorlib.dll*, welche die Basistypen des .NET Frameworks in verschiedenen Namensräumen umfasst (siehe obige Tabelle).

Zugriff auf COM-Komponenten

Verweise auf COM-DLLs werden so eingebunden, dass sie zur Entwurfszeit quasi wie .NET-Komponenten behandelt werden können.

Über das Menü *Projekt|Verweis hinzufügen...* und Auswahl des "COM"-Registers erreichen Sie die Liste der verfügbaren COM-Bibliotheken. Nachdem Sie die gewünschte Bibliothek selektiert haben, können Sie die COM-Komponente wie gewohnt ansprechen.

HINWEIS: Wenn Sie COM-Objekte, wie z.B. alte ADO-Bibliotheken, in Ihre .NET-Projekte einbinden wollen, müssen Sie auf viele Vorteile von .NET verzichten. Durch die zusätzliche Interoperabilitätsschicht sinkt die Performance meist deutlich.

Metadaten und Reflexion

Das .NET-Framework stellt im *System.Reflection*-Namespace einige Klassen bereit, die es erlauben, die Metadaten (Beschreibung bzw. Strukturinformationen) einer Assembly zur Laufzeit auszuwerten, womit z.B. eine Untersuchung aller dort enthaltenen Typen oder Methoden möglich ist.

Die Beschreibung durch die .NET-Metadaten ist allerdings wesentlich umfassender als es in den gewohnten COM-Typbibliotheken üblich war. Außerdem werden die Metadaten direkt in der Assembly untergebracht, die dadurch selbstbeschreibend wird und z.B. auf Registry-Einträge verzichten kann. Metadaten können daher nicht versehentlich verloren gehen oder mit einer falschen Dateiversion kombiniert werden.

HINWEIS: Es gibt unter .NET nur noch eine einzige Stelle, an der sowohl der Programmcode als auch seine Beschreibung gespeichert wird!

Metadaten ermöglichen es, zur Laufzeit festzustellen, welche Typen benutzt und welche Methoden aufgerufen werden. Daher kann .NET die Umgebung an die Anwendung anpassen, sodass diese effizienter arbeitet.

Der Mechanismus zur Abfrage der Metadaten wird Reflexion (*Reflection*) genannt. Das .NET-Framework bietet dazu eine ganze Reihe von Methoden an, mit denen jede Anwendung – nicht nur die CLR – die Metadaten von anderen Anwendungen abfragen kann.

Auch Entwicklungswerkzeuge wie Microsoft Visual Studio verwenden die Reflexion, um z.B. den Mechanismus der IntelliSense zu implementieren. Sobald Sie einen Methodennamen eintippen, zeigt die IntelliSense eine Liste mit den Parametern der Methode an oder mit allen Elementen eines bestimmten Typs.

Weitere nützliche Werkzeuge, die auf der Basis von Reflexionsmethoden arbeiten, sind der IL-Disassembler (ILDASM) des .NET Frameworks oder ILSpy.

HINWEIS: Eine besondere Bedeutung hat Reflexion im Zusammenhang mit dem Auswerten von Attributen zur Laufzeit (siehe folgender Abschnitt).

Attribute

Wer sich noch an die älteren objektorientierten Sprachen (VB 6, Delphi 7, ...) erinnert, der kennt Attribute als Variablen, die zu einem Objekt gehören und damit seinen Zustand beschreiben.

Unter .NET haben Attribute eine grundsätzlich andere Bedeutung:

HINWEIS: .NET-Attribute stellen einen Mechanismus dar, mit welchem man Typen und Elemente einer Klasse schon beim Entwurf kommentieren und mit Informationen versorgen kann, die sich zur Laufzeit mittels Reflexion abfragen lassen.

Auf diese Weise können Sie eigenständige selbstbeschreibende Komponenten entwickeln, ohne die erforderlichen Infos separat in Ressourcendateien oder Konstanten unterbringen zu müssen. So erhalten Sie mobilere Komponenten mit besserer Wartbarkeit und Erweiterbarkeit.

Man kann Attribute auch mit "Anmerkungen" vergleichen, die man einzelnen Quellcode-Elementen, wie Klassen oder Methoden, "anheftet". Solche Attribute gibt es eigentlich in jeder Programmiersprache, sie regeln z.B. die Sichtbarkeit eines bestimmten Datentyps. Allerdings waren diese Fähigkeiten bislang fest in den Compiler integriert, während sie unter .NET nunmehr direkt im Quellcode zugänglich sind. Das heißt, dass .NET-Attribute typsichere, erweiterbare Metadaten sind, die zur Laufzeit von der CLR (oder von beliebigen .NET-Anwendungen) ausgewertet werden können.

Mit Attributen können Sie Design-Informationen definieren (z.B. zur Dokumentation), Laufzeit-Infos (z.B. Namen einer Datenbankspalte für ein Feld) oder sogar Verhaltensvorschriften für

die Laufzeit (z.B. ob ein gegebenes Feld an einer Transaktion teilnehmen darf). Die Möglichkeiten sind quasi unbegrenzt.

Wenn Ihre Anwendung beispielsweise einen Teil der erforderlichen Informationen in der Registry abspeichert, muss bereits beim Entwurf festgelegt werden, wo die Registrierschlüssel abzulegen sind. Solche Informationen werden üblicherweise in Konstanten oder in einer Ressourcendatei untergebracht oder sogar fest in die Aufrufe der entsprechenden Registrierfunktionen eingebaut. Wesentliche Bestandteile der Klasse werden also von der übrigen Klassendefinition abgetrennt. Der Attribute-Mechanismus macht damit Schluss, denn er erlaubt es, derlei Informationen direkt an die Klassenelemente "anzuheften", so dass letztendlich eine sich vollständig selbst beschreibende Komponente vorliegt.

Serialisierung

Fester Bestandteil des .NET-Frameworks ist auch ein Mechanismus zur Serialisierung von Objekten. Unter Serialisierung versteht man das Umwandeln einer Objektinstanz in sequenzielle Daten, d.h. in binäre oder XML-Daten oder in eine SOAP-Nachricht mit dem Ziel, die Objekte als Datei permanent zu speichern oder über Netzwerke zu verschicken.

Auf umgekehrtem Weg rekonstruiert die Deserialisierung aus den Daten wieder die ursprüngliche Objektinstanz.

Das .NET-Framework unterstützt zwei verschiedene Serialisierungsmechanismen:

- Die *Shallow-Serialisierung* mit der Klasse *System.Xml.Serialization.XmlSerializer*.
- Die *Deep-Serialisierung* mit den Klassen *BinaryFormatter* und *SoapFormatter* aus dem *System.Runtime.Serialization*-Namespace.

Aufgrund ihrer Einschränkungen (geschützte und private Objektfelder bleiben unberücksichtigt) ist die Shallow-Serialisierung für uns weniger interessant. Hingegen werden bei der Deep-Serialisierung alle Felder berücksichtigt, Bedingung ist lediglich die Kennzeichnung der Klasse mit dem Attribut *[Serializable]*.

Anwendungsgebiete der Serialisierung finden sich bei ASP.NET, ADO.NET, XML etc.

Multithreading

Multithreading ermöglicht es einer Anwendung, ihre Aktivitäten so aufzuteilen, dass diese unabhängig voneinander ausgeführt werden können, bei gleichzeitig besserer Auslastung der Prozessorzeit. Allgemein sind Threads keine Besonderheit von .NET, sondern auch in anderen Programmierumgebungen durchaus üblich.

Unter .NET laufen Threads in einem Umfeld, das Anwendungsdomäne genannt wird, Erstellung und Einsatz erfolgen mit Hilfe der Klasse *System.Threading.Thread*.

Nicht in jedem Fall ist die Aufnahme zusätzlicher Threads die beste Lösung, da man sich dadurch auch zusätzliche Probleme einhandeln kann. So ist beim Umgang mit mehreren Threads die Threadsicherheit von größter Bedeutung, d.h., aus Sicht der Threads müssen die Objekte stets in

einem gültigen Zustand vorliegen und das auch dann, wenn sie von mehreren Threads gleichzeitig benutzt werden.

Objektorientierte Programmierung

Last, but not least, wollen wir am Ende unseres Rundflugs über die .NET-Higlights noch einmal auf das allem zugrunde liegende OOP-Konzept verweisen, denn .NET ist komplett objektorientiert aufgebaut – unabhängig von der verwendeten Sprache oder der Zielumgebung, für die programmiert wird (Windows- oder Web-Anwendung).

Jeder .NET-Code ist innerhalb einer Klasse verborgen, und sogar einfache Variablen sind zu Objekten mutiert, die Eigenschaften und Methoden bereitstellen. Es macht deshalb wenig Sinn, mit der Einführung in die Sprache C# fortzufahren ohne sich vorher mit dem Konzept der OOP vertraut gemacht zu haben (siehe Kapitel 3).

1.6 Wichtige Neuigkeiten in Visual Studio 2012

All unseren Lesern, die bereits mit der Vorgängerversion (Visual Studio 2010, C# 4.x) gearbeitet haben, sind wir einen kleinen Überblick über die wichtigsten Neuerungen der Version 2012 (bzw. C# 5.0) schuldig.

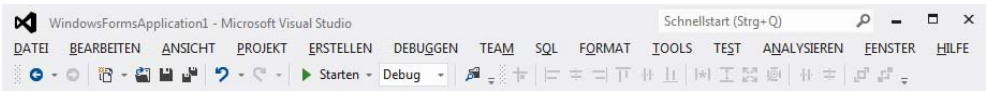
1.6.1 Die neue Visual Studio 2012 Entwicklungsumgebung

Das User Interface von Visual Studio 2012 wurde, aus welchen Gründen auch immer, deutlich umgestaltet und ist – unter weitgehendem Verzicht auf 3D-Effekte – vornehmlich in tristes Grau gehüllt[1]. Der Umsteiger wird einige Zeit brauchen, um altbekannte Funktionen an anderer Stelle wiederzufinden.

Neues Outfit der Toolbar

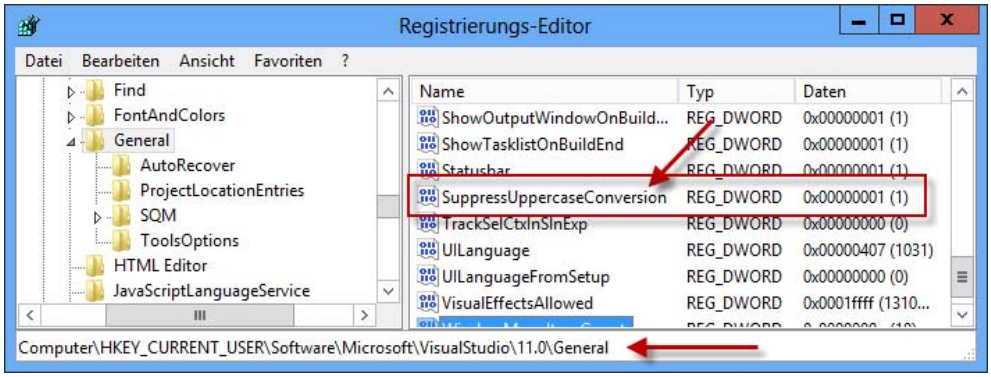
Die neue Schnellstart-Box (oben rechts) soll die die bequeme Suche nach momentan verfügbaren Befehlen und deren Auswahl in einer Dropdown-Liste ermöglichen.

Sehr gewöhnungsbedürftig ist die komplette Großschreibung der Menü-Oberpunkte:

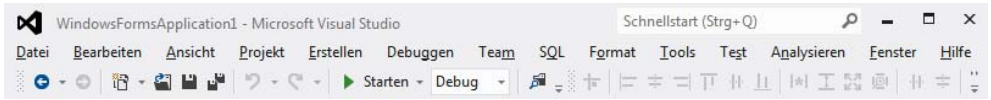


Wem dieser aufdringliche Stil nicht gefällt, für den gibt es eine Lösung: Rufen Sie die Registry auf (*regedit* im Suchfeld des Windows-Startmenüs eingeben). Unter dem Schlüssel *HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\11.0\General* erzeugen Sie einen neuen DWORD-Eintrag mit dem Namen *SuppressUppercaseConversion* und weisen diesem den Wert 1 zu.

[1] Die Alternative, die Sie unter *Tools|Optionen* wählen können, wäre eine gruselige schwarze Bedienoberfläche.

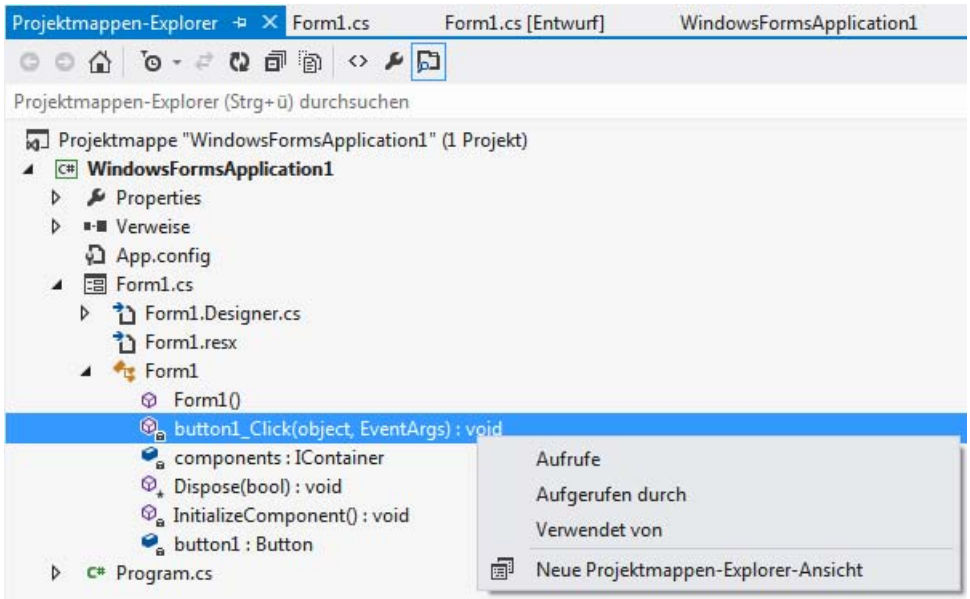


Nach dem Neustart von Visual Studio werden Sie ab jetzt von einem viel freundlicheren Menü begrüßt:



Veränderter Projektmappen-Explorer

Der Projektmappen-Explorer ist zum "Mädchen für alles" mutiert und unterstützt jetzt auch die Navigation durch das Objektmodell, Volltextsuche und mehr. Zum Beispiel können Sie eine *.cs*-Datei expandieren, um die Klassen innerhalb der Datei zu betrachten, dann die Klasse weiter expandieren, um ihre Mitglieder und deren Aufrufhierarchien zu untersuchen (siehe Abbildung).



Registerkartenverwaltung der geöffneten Dateien

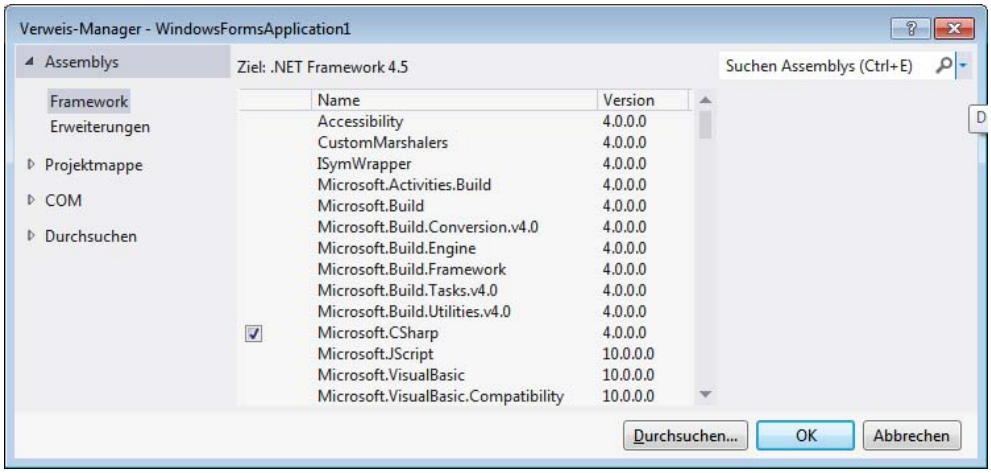
Hier sollten Sie sich an eine interessante Neuigkeit gewöhnen: Haben Sie bisher eine Datei per Doppelklick im Projektmappen-Explorer geöffnet, so gibt es nunmehr die zusätzliche Möglichkeit, die Datei per Einfachklick als Vorschau temporär zu öffnen.

Suchen überall

In Visual Studio 2012 wird jetzt die (mehr oder weniger sinnvolle) Suche nach zahlreichen Features an zahlreichen Orten fast bis zur Perversion ausgeweitet: Projektmappen-Explorer, Verweis hinzufügen, Integrierte Schnellsuche, Test Explorer, Fehlerliste, Parallel Watch, Werkzeugkasten, Team Foundation Server (TFS) u.a.

Neuer Verweis Manager-Dialog

Der Dialog *Verweis-Manager* wurde bereits in den Power Tools für Visual Studio 2010 erneuert, jetzt findet man ihn fest integriert in Visual Studio 2012. Der sich öffnende Verweis-Manager bietet eine Übersicht über .NET-Komponenten im Global Assembly Cache und in den in der Registry gespeicherten Suchpfaden, zu Projekten in der gleichen Projektmappe und COM-Komponenten. Enthalten ist auch ein Browser zur Suche nach Assemblies.



Projekt-Kompatibilität

Visual Studio 2012 enthält jetzt die Projekt-Kompatibilität als spezielles Feature, sodass ein Projekt-Upgrade als gemeinsamer Schritt im Team nicht erforderlich ist. Projektdateien, die unter Visual Studio 2010 erzeugt wurden, bleiben auch nach ihrem Öffnen in Visual Studio 2012 unverändert. Wenn also ein Entwickler im Team ein Visual Studio 2010 Projekt in Visual Studio 2012 öffnet und den gemeinsamen Code ändert, können andere Entwickler dasselbe Projekt unter Visual Studio 2010 SP1 öffnen. Umgekehrt können auch Entwickler für Visual Studio 2012-Projekte daran gemeinsam mit anderen Entwicklern arbeiten, die Visual Studio 2010 SP1 verwenden.

HINWEIS: Die Projekt-Kompatibilität in Visual Studio 2012 funktioniert nur mit Visual Studio 2010 SP1, alternativ werden Sie aufgefordert das Projekt zu konvertieren.

Neue Projekttypen

Die meisten neuen Projekttypen gibt es bei den neuen Windows Store-Anwendungen und bei JavaScript, sie sind allerdings nur verfügbar, wenn Visual Studio 2012 auf einem Windows 8-Computer läuft. Diese Applikationen benötigen die Windows Runtime (WinRT) und können in C#, Visual Basic, C++ und JavaScript entwickelt werden. Achten Sie bei den einzelnen Projekttypen immer auch auf die jeweils unterstützte Framework-Version:

HINWEIS: Visual Studio 2012 kann bis zurück zur Version 2.0 kompilieren. Bei älteren Versionen fehlen dann allerdings viele Projekt-Vorlagen.

Multi-Monitor -Unterstützung

Visual Studio 2012 hat jetzt einen stark verbesserten Multi-Monitor-Support. Auf elegante Weise ist es möglich, die IDE auf mehrere Monitore zu verteilen.

Zusätzliche Tools und Features

Hervorzuheben ist die Integration von Expression Blend in die Visual Studio 2012 IDE. Auch Visual Studio LightSwitch, die Entwicklungsumgebung für das Rapid Application Development (RAD), und das Application Lifecycle Management (ALM) gehören jetzt dazu.

1.6.2 Neuheiten im .NET Framework 4.5

Auch hier wollen wir nur die unserer Meinung nach wichtigsten Features hervorheben:

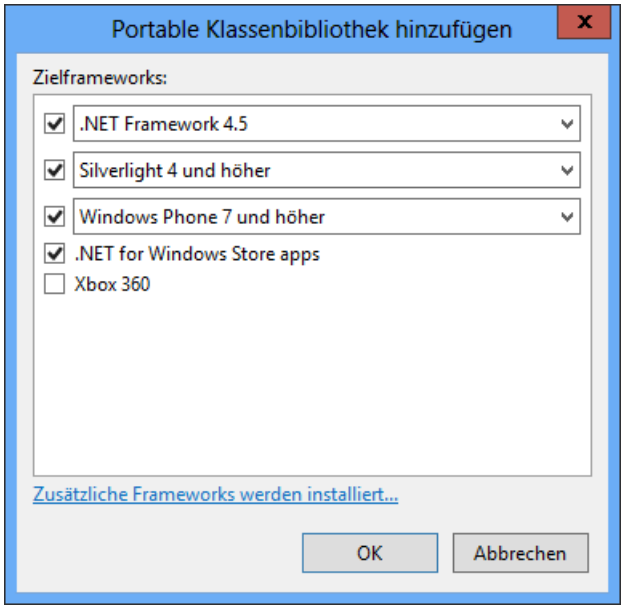
WinRT-Anwendungen

Unter Windows 8 kann eine Teilmenge von .NET Framework 4.5 zum Erstellen von Windows-Apps im WinRT-Stil verwendet werden (siehe Kapitel 17 bis 21).

Mit dem *Resource File Generator*-Tool (*Resgen.exe*) können Sie aus einer RESOURCES-Datei, die in einer .NET Framework-Assembly eingebettet ist, eine RESW-Datei für Windows-Apps im WinRT-Stil erstellen.

Portable Klassenbibliotheken

Ein *Portable Klassenbibliothek*-Projekt erlaubt das Erstellen von verwalteten Assemblies für mehrere .NET-Framework-Plattformen. Nachdem Sie sich für die Zielplattform (Windows Phone, .NET für Windows Store-Apps) entschieden haben, werden die verfügbaren Typen und Mitglieder automatisch auf diese Plattformen beschränkt.



Parallele Computervorgänge

Das Framework 4.5 enthält mehrere neue Funktionen und Erweiterungen für parallele Berechnungen. Dazu gehören die verbesserte Unterstützung für asynchrone Programmierung, die optimierte Unterstützung für paralleles Debuggen und eine neue Datenflussbibliothek.

Internet

Hinzugekommen sind einige neue Funktionen für ASP.NET 4.5: Unterstützung für neue HTML5-Formulartypen, für das asynchrone Lesen und Schreiben von HTTP-Anforderungen und -Antworten, für asynchrone Module, für die E-Mail-Adressen-Internationalisierung (EAI) und für das WebSockets-Protokoll.

WPF

Mit dem *ibbon*-Steuerelement können Sie eine Menüband-Benutzeroberfläche programmieren, die ein Anwendungsmenü und eine Symbolleiste für den Schnellzugriff enthält.

Die synchrone und asynchrone Datenvalidierung wird durch eine neue *INotifyDataErrorInfo*-Schnittstelle unterstützt.

Auch die Datenbindung an statische Eigenschaften und an benutzerdefinierte Typen, die die *ICustomTypeProvider*-Schnittstelle implementieren, ist möglich geworden.

WCF

Hervorzuheben ist hier vor allem die Unterstützung für die Contract-First-Entwicklung sowie für asynchrones Streaming.

1.6.3 C# 5.0 – Sprache und Compiler

Wie die folgende Tabelle zeigt, sind die Neuerungen im Vergleich zu denen der Vorgängerversionen relativ bescheiden.

Version	IDE	Wichtigste Neuerungen					
C# 1.0	VS 2002	Managed Code					
C# 2.0	VS 2005	Generics	Anonyme Methoden	Nullable Types			
C# 3.0	VS 2008	Lambda Ausdrücke	Erweiterungs-methoden	Expression Tree	Anonyme Typen	LINQ	Typinferenz (var)
C# 4.0	VS 2010	Late Binding (dynamisch)	Benannte Argumente	Optionale Parameter	Mehr COM-Support	Parallele Programmierung (TPL) PLINQ	
C# 5.0	VS 2012	Asynchrone Features	Caller Information				

Asynchrone Methoden

Im Zusammenhang mit der asynchronen Programmierung wurden zwei neue Schlüsselwörter eingeführt: der *async*-Modifizierer und der *await*-Operator. Eine mit *async* markierte Methode heißt "asynchrone Methode". Es ergeben sich dadurch teilweise erhebliche Vereinfachungen für den Programmierer (siehe Abschnitt 9.9).

Caller Information

Dieses neue Feature kann hilfreich sein beim Debugging und beim Entwickeln von Diagnose-Tools. So kann doppelter Code vermieden werden, wie zum Beispiel beim Logging und Tracing (siehe Abschnitt 11.3).

1.7 Praxisbeispiele

Bereits im Abschnitt 1.3.3 hatten wir Ihnen die vier Etappen der Programmentwicklung in Visual Studio ganz allgemein erklärt. Jetzt wollen wir Nägel mit Köpfen machen und diese Schritte anhand von zwei Beispielen (ein ganz einfaches und ein etwas anspruchsvolleres) praktisch nachvollziehen.

Für diese kleinen Applikationen sind nicht die geringsten Programmierkenntnisse erforderlich, es geht vielmehr darum, ein erstes Gefühl für die Anwendungsentwicklung unter Visual Studio 2012 zu gewinnen.

1.7.1 Windows Forms-Anwendung für Einsteiger

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über welche per Mausklick die Beschriftung der Titelleiste in "Hallo C#-Freunde" geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene

Windows-Anwendungen erstellen kann. Der damit ausgelöste Aha-Effekt wird Sie sicher ausreichend motivieren, manche Durststrecken der nächsten Kapitel zu überstehen.

1. Etappe: Visueller Entwurf der Bedienoberfläche

Der Programmstart von *Microsoft Visual Studio 2012* erfolgt entweder über das Windows-Startmenü oder noch schneller über eine Desktop- bzw. Taskleisten-Verknüpfung.

Auf der Startseite klicken Sie den Link *Neues Projekt...*. Im sich daraufhin öffnenden Dialogfenster *Neues Projekt* wählen Sie links in der Baumstruktur unter *Vorlagen* zunächst *Visual C#* und *Windows* aus (siehe Abbildung Seite 69).

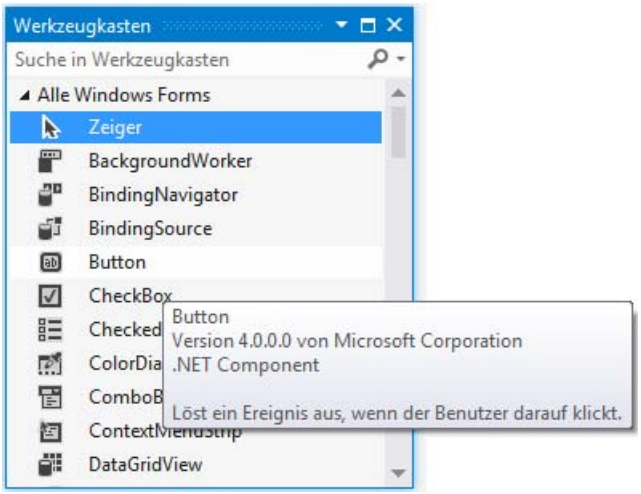
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

Name: *WindowsFormsApplication1*
Ort: z.B.: *C:\CS\Beispiele*
Projektmappenname: *WindowsFormsApplication1*

HINWEIS: Im Dialogfenster rechts oben sehen Sie, dass Sie mit Visual Studio 2012 sowohl Projekte für das .NET Framework 4.5 als auch für die Vorgängerversionen 4, 3.5, 3.0 und 2.0 entwickeln können. Entsprechend der eingestellten Version ändert sich auch das Vorlagen-Angebot.

Nach dem "OK" dauert es ein kleines Weilchen, bis die Entwicklungsumgebung mit dem Startformular *Form1* erscheint. Darauf platzieren Sie ein Steuerelement vom Typ *Button*. Die dazu notwendige Vorgehensweise unterscheidet sich kaum von der bei einem normalen Zeichenprogramm.

Klicken Sie im Menü *Ansicht* auf den Eintrag *Werkzeugkasten* und wählen Sie dann einfach das gewünschte Steuerelement aus:



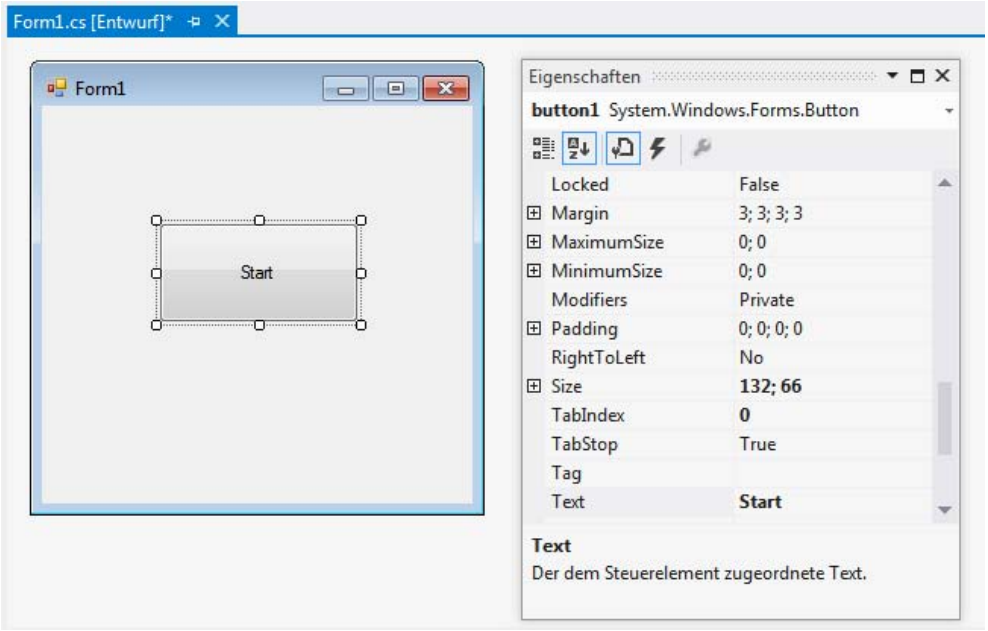
Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* in gewohnter Windows-Manier auf das Formular ziehen, dort absetzen und auf die gewünschte Größe zoomen.

2. Etappe: Zuweisen der Objekteigenschaften

Der *Button* trägt noch seine standardmäßige Beschriftung *button1*. Um diese in "Start" zu ändern, muss die *Text*-Eigenschaft geändert werden. Markieren Sie dazu das Steuerelement mit der Maus und rufen Sie mit F4 (bzw. über das Menü *Ansicht*) das Eigenschaftenfenster auf. Ändern Sie im Eigenschaften-Fenster die *Text*-Eigenschaft von ihrem Standardwert "button1" in "Start".

Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Immer wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.

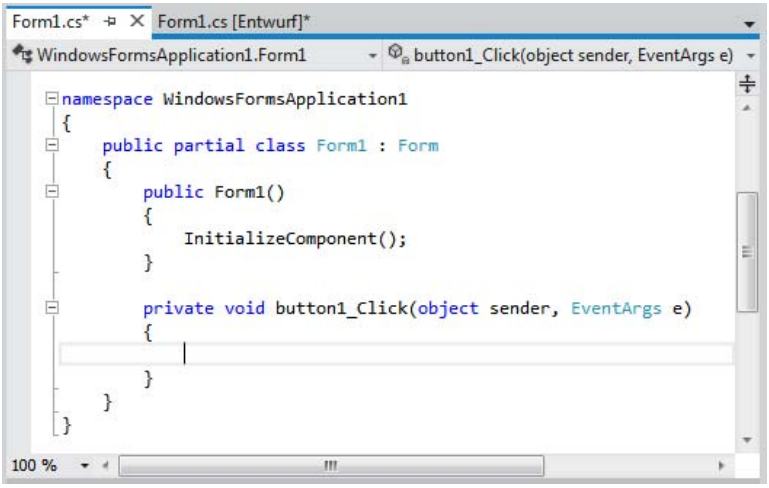
Es dürfte Ihnen nun auch keine Schwierigkeiten bereiten, über die *Font*-Eigenschaft von *button1* auch noch die Schriftgröße etc. zu ändern.



3. Etappe: Verknüpfen der Objekte mit Ereignissen

Klicken Sie doppelt auf die Komponente *button1*, so öffnet sich das Code-Fenster. Richten Sie nun Ihr Augenmerk auf die Schreibmarke, welche im vorgefertigten Rahmencode innerhalb der *Click*-Ereignisbehandlungsroutine (Eventhandler) blinkt. Hier tragen Sie Ihren C#-Code ein, der festlegt, **was** passieren soll, wenn zur Programmlaufzeit (also nicht jetzt zur Entwurfszeit!) der Anwender auf diese Schaltfläche klickt.

In unserem Fall wollen wir erreichen, dass sich die Beschriftung der Titelleiste des Formulars ändert. Das bedeutet, dass wir die *Text*-Eigenschaft des Objekts *Form1*, dessen Standardwert bislang ebenfalls "Form1" lautete, neu zuweisen müssen. Diesmal aber tun wir das nicht im Eigenschaftenfenster, sondern per C#-Code.



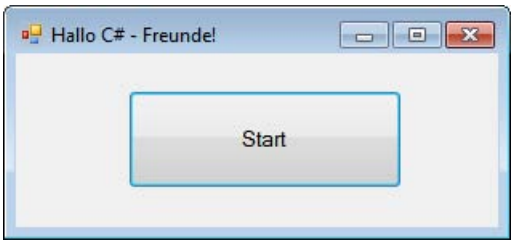
Fügen Sie die fett hervorgehobene Zeile in den bereits vorhandenen Rahmencode ein:

```
private void button1_Click(object sender, EventArgs e)
{
    this.Text = "Hallo C# - Freunde!";
}
```

4. Etappe: Programm kompilieren und testen

Kompilieren Sie das Programm durch Klicken auf das kleine Dreieck in der Symbolleiste (bzw. Menü *Debuggen|Debugging starten* oder *F5*). Sie befinden sich nun im Ausführungsmodus. Ihr Programm "lebt" jetzt, denn die Schaltfläche lässt sich klicken, und die Beschriftung der Titelleiste ändert sich tatsächlich.

Das Programm beenden Sie, indem Sie auf das kleine Quadrat in der Symbolleiste klicken (bzw. Menü *Debuggen/Debugging beenden* oder *Umschalt+F5*) oder aber das Formular einfach in altbekannter Windows-Manier schließen.



Gratulation – Sie haben soeben Ihre erste Windows Forms-Anwendung geschrieben!

Bemerkungen

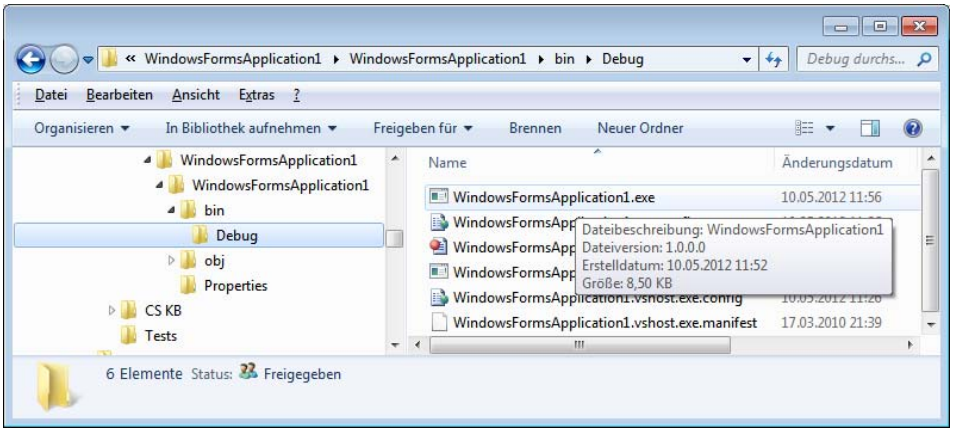
- In diesem Beispiel haben Sie ganz nebenbei auch gelernt, dass man Eigenschaften (Properties) nicht nur zur Entwurfszeit im Eigenschaften-Fenster zuweist, sondern dies auch zur Laufzeit per Code tun kann. Im letzteren Fall wird der Name der Eigenschaft (*Text*) vom zugehörigen Objekt (*this*) durch einen Punkt getrennt.
- Die Properties, die Sie im Eigenschaften-Fenster zuweisen, bezeichnet man auch als *Start-eigenschaften*. Zur Laufzeit können diese – wie im Beispiel für die *Text*-Eigenschaft des Formulars gezeigt – durchaus ihren Wert ändern.
- Das .NET-SDK empfiehlt, dass alle Klassen innerhalb eines Namensraums (*namespace*) definiert werden[1]. Visual Studio verwendet automatisch den Namen Ihres Projekts (wir haben das bei der Standardvorgabe *WindowsFormsApplication1* belassen) als oberste Ebene des Namensraums. Davor wurden über den *using*-Befehl automatisch weitere Namensräume standardmäßig eingebunden. Insgesamt hat also der Quellcode Ihrer ersten Windows-Anwendung folgendes Aussehen, wobei nur die fett hervorgehobene Zeile von Ihnen selbst getippt werden musste:

```
using System;
...
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Text = "Hallo C# - Freunde!";
        }
    }
}
```

HINWEIS: Beachten Sie die durch die {}-Klammern eingegrenzten Gültigkeitsbereiche!

[1] Mehr zum Konzept der Namensräume erfahren Sie im Kapitel 5.

- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie im Unterverzeichnis *...\WindowsFormsApplikation1\bin\Debug* Ihres Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische Exe-Datei, sondern um eine so genannte *Assemblierung* (siehe Abschnitt 1.5). Da die EXE im MSIL-Code vorliegt, ist sie nur in Zusammenarbeit mit dem .NET-Framework lauffähig. Haben Sie die Programmentwicklung erfolgreich abgeschlossen und Visual Studio beendet, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsFormsApplication1.exe* das Programm zu starten.



Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsFormsApplication1.sln*. Wenn Sie auf diese Datei klicken[1], so wird standardmäßig Visual Studio geöffnet und das Programm wird in die Entwicklungsumgebung geladen.

1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger

Diesmal soll es keine Spielerei, sondern ein durchaus nützliches Prögrämmchen sein – die Umrechnung von Euro in Dollar, ein simpler Währungsrechner also. Durch Vergleichen mit der im Abschnitt 1.2 beschriebenen ersten C#-Anwendung dürften auch die Unterschiede der klassischen Konsolentechnik zur visualisierten, objekt- und ereignisorientierten Windows-Programmierung ganz deutlich zu Tage treten.

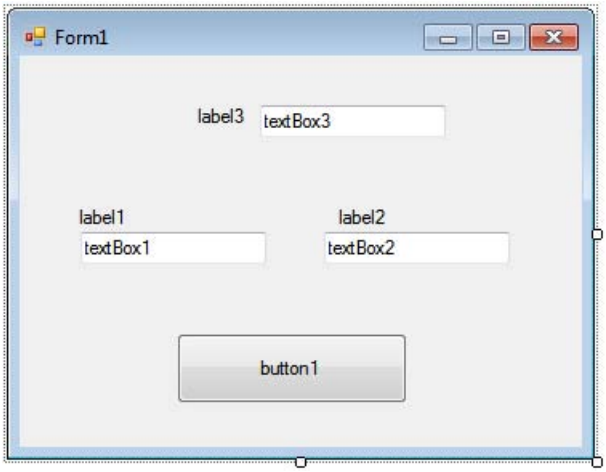
1. Etappe: Visueller Entwurf der Bedienoberfläche

Öffnen Sie ein neues Visual C#-Projekt vom Typ "Windows Forms-Anwendung" und geben Sie ihm den Namen "EuroDollar".

Ziel ist die folgende Bedienoberfläche, die Sie jetzt mühelos im Designer-Fenster erstellen (siehe folgende Abbildung)[2].

[1] Das werden Sie z B. häufig beim Laden von Beispielprojekten tun.

[2] Der Inhalt der drei Textboxen ist standardmäßig leer und wurde nur hier aus Übersichtlichkeitsgründen mit deren Namen beschriftet.



Sie brauchen außer dem bereits vorhandenen Startformular *Form1* drei *Label* zur Beschriftung, drei *TextBox*en für die Eingabe und einen *Button* zum Beenden des Programms. Für die Namensgebung sorgt Visual Studio automatisch, es sei denn, Sie möchten den Objekten eigene Namen verleihen.

HINWEIS: Konzentrieren Sie sich in der ersten Etappe nur auf Lage und Abmessung der Komponenten, nicht auf deren Beschriftung, da Eigenschaften erst in der nächsten Etappe angepasst werden!

Beim Platzieren und bei der Größenanpassung der Komponenten gehen Sie ähnlich vor, wie Sie es bereits von vektororientierten Zeichenprogrammen (Visio, PowerPoint) gewöhnt sind:

- Im Werkzeugkasten klicken Sie auf das Symbol für die *TextBox*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *textBox1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) die *TextBox* auf ihre endgültige Größe. Analog verfahren Sie mit *textBox2* und *textBox3*.
- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *Label*-Komponente und erzeugen auf die gleiche Weise *label1*, *label2* und *label3*.
- Schließlich bleibt noch *button1*, den Sie am unteren Rand von *Form1* absetzen.

2. Etappe: Zuweisen der Objekteigenschaften

Unser Programm besteht nun aus insgesamt acht Komponenten: einem Formular und sieben Steuerelementen. Alle Eigenschaften haben bereits ihre Standardwerte. Einige davon müssen wir allerdings noch ändern. Dies geschieht mit Hilfe des Eigenschaften-Fensters. Wenn Sie mit der

Maus auf eine Komponente klicken und danach die *F4*-Taste betätigen, erscheint das Eigenschaften-Fenster der Komponente mit der Liste aller zur Entwurfszeit verfügbaren Eigenschaften.

- Beginnen Sie mit *label1*, das die Beschriftung "Euro" tragen soll. Die Beschriftung kann mit der *Text*-Eigenschaft geändert werden. Standardmäßig entspricht diese der *Name*-Property, in unserem Fall also "*label1*". Um das zu ändern, klicken Sie auf das *Label* und tragen anschließend in der Spalte rechts neben dem *Text*-Feld die neue Beschriftung ein (die alte ist vorher "wegzuradieren"). Analog verfahren Sie mit den beiden anderen *Label*s (Beschriftung "Dollar" und "Kurs 1: ").
- Auch *button1* muss natürlich seine neue *Text*-Eigenschaft ("Beenden") erhalten.
- Schließlich klicken Sie auf eine leere Fläche von *Form1*, um anschließend mit *F4* das Eigenschaften-Fenster für das Formular aufzurufen und dessen *Text*-Eigenschaft entsprechend der gewünschten Beschriftung der Titelleiste zu modifizieren.

Die Tabelle gibt eine Zusammenstellung aller Objekteigenschaften, die wir geändert haben:

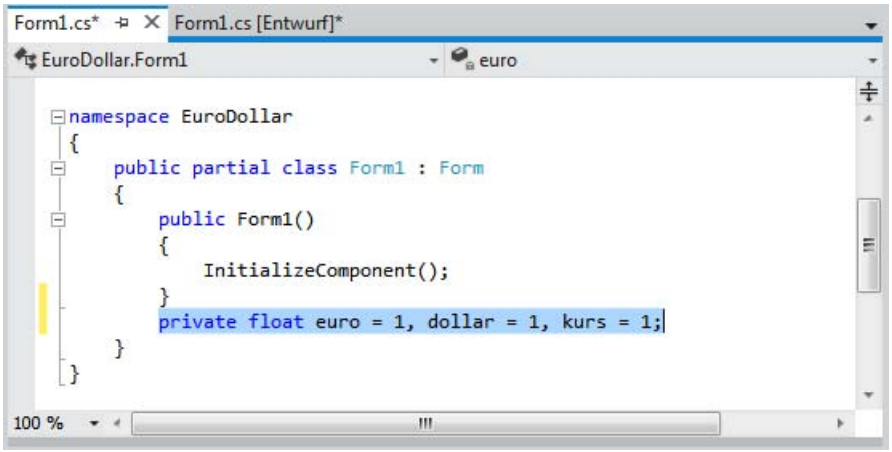
Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Währungsrechner *10*
label1	*Text*	Euro
label2	*Text*	Dollar
label3	*Text*	Kurs 1:
textBox1	*TextAlign*	*Right*
textBox2	*TextAlign*	*Right*
textBox3	*TextAlign*	*Center*
button1	*Text*	Beenden

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Während Sie die beiden Vorgängeretappen noch getrost Ihrer Sekretärin überlassen konnten, beginnt jetzt Ihre Hauptarbeit als C#-Programmierer. Wechseln Sie zum Code-Fenster *Form1.cs* (auch mit *F7*, *Ansicht|Code* oder dem Kontextmenü des Formulars möglich). Was Sie erwartet, ist die von Visual Studio vorbereitete Klassendeklaration von *Form1*. Wie Sie sehen, können Sie einzelne Bereiche (Regionen) durch das Plus- bzw. Minus-Symbol am linken Rand auf- bzw. zuklappen.

Zunächst fügen Sie eine Anweisung ein, um drei Variablen des *float*-Datentyps zu deklarieren. Gleichzeitig werden diese Variablen mit dem Wert 1 initialisiert:

```
private float euro=1, dollar=1, kurs=1;
```

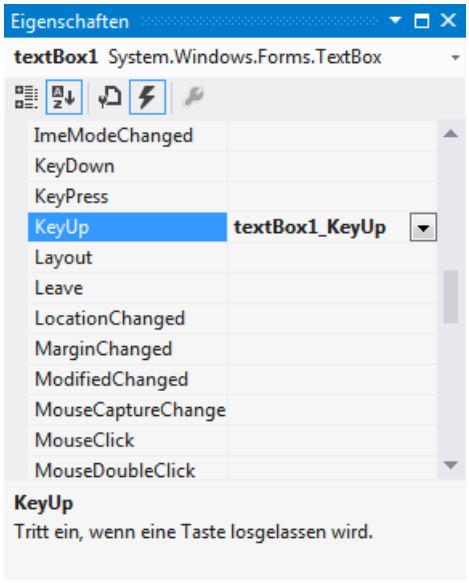


Im Unterschied zur einfachen Konsolenanwendung, bei welcher uns das Programm die Einhaltung einer bestimmten Eingabereihenfolge aufgezwungen hat, soll in unserer Windows Forms-Anwendung die Berechnung immer dann neu gestartet werden, wenn wir bei der Eingabe in eine der drei Textboxen irgendeine Taste losgelassen haben. Wir müssen also für jede der Textboxen einen eigenen Eventhandler für das *KeyUp*-Ereignis schreiben!

Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, die Sie mit fortschreitender Programmierpraxis sehr bald auch im Schlaf ausführen können:

- **Objekt auswählen**
  Zur Objektauswahl klicken Sie auf das Objekt im Designer-Fenster und öffnen mit *F4* das Eigenschaften-Fenster.
  Klicken Sie im Eigenschaften-Fenster oben auf das ⚡-Symbol, um die Ereignisliste zur Anzeige zu bringen.
- **Ereignis auswählen**
  Zur Ereignisauswahl doppelklicken Sie auf das gewünschte Ereignis. Als Resultat werden automatisch die erste und die letzte Zeile (Rahmencode) des Eventhandlers generiert und im Code-Fenster angezeigt.
- **Ereignisbehandlungen programmieren**
  Füllen Sie den Eventhandler mit den gewünschten C#-Anweisungen aus.

Wir beginnen in unserem Beispiel mit dem *KeyUp*-Eventhandler für *textBox1*, der immer dann ausgeführt wird, wenn der Benutzer den Euro-Betrag ändert. Doppelklicken Sie also auf *KeyUp* und das Codefenster öffnet sich mit dem automatisch erzeugten Rahmencode des Eventhandlers (siehe folgende Abbildung).



Füllen Sie nun den Rahmencode wie folgt aus:

```
private void textBox1_KeyUp(object sender, KeyEventArgs e)
{
    euro = Convert.ToSingle(textBox1.Text);
    dollar = euro * kurs;
    textBox2.Text = dollar.ToString("#,##0.00");
}
```

HINWEIS: Sie müssen nur die Anweisungen innerhalb der geschweiften Klammern selbst einfügen, der übrige Rahmencode wird automatisch erzeugt, wenn Sie die oben erläuterte Erstellungsreihenfolge beachten!

Der Prozedurkopf des Eventhandlers verweist standardmäßig vor dem Unterstrich (_) auf den Namen des Objekts und danach auf das entsprechende Ereignis. Das vorangestellte *private* verdeutlicht, dass auf die Prozedur nur innerhalb des *Form1*-Klasse zugegriffen werden kann.

Auf analoge Weise erzeugen Sie die Eventhandler für die Steuerelemente *textBox2* und *textBox3*, geben aber dann den jeweils geänderten Umrechnungscode ein.

Ändern des Dollar-Betrags:

```
private void textBox2_KeyUp(object sender, KeyEventArgs e)
{
     dollar = Convert.ToSingle(textBox2.Text);
     euro = dollar / kurs;
     textBox1.Text = euro.ToString("#,##0.00");
}
```

Ändern des Umrechnungskurses:

```
private void textBox3_KeyUp(object sender, KeyEventArgs e)
{
     kurs = Convert.ToSingle(textBox3.Text);
     dollar = euro * kurs;
     textBox2.Text = dollar.ToString("#,##0.00");
}
```

HINWEIS: Grübeln Sie jetzt noch nicht über den tieferen Sinn der einzelnen Anweisungen nach, denn dazu haben Sie in den späteren Kapiteln noch genug Zeit!

Damit Sie bereits unmittelbar nach dem Programmstart sinnvolle Werte in den drei Textboxen sehen, ist folgender Eventhandler für das *Load*-Ereignis des *Form1*-Objekts hinzuzufügen:

```
private void Form1_Load(object sender, EventArgs e)
{
     textBox1.Text = euro.ToString();
     textBox2.Text = dollar.ToString();
     textBox3.Text = kurs.ToString();
}
```

Beim Klick auf den *Beenden*-Button soll das Formular entladen werden. Wählen Sie in der Objektauswahlliste des Eigenschaften-Fensters den Eintrag *button1* und anschließend in der Ereignisauswahlliste das *Click*-Event:

```
private void button1_Click(object sender, EventArgs e)
{
     this.Close();
}
```

4. Etappe: Programm kompilieren und testen

Klicken Sie auf den ▶ Starten-Button in der Symbolleiste (oder *F5*-Taste), und im Handumdrehen ist Ihre C#-Windows-Anwendung kompiliert und ausgeführt!

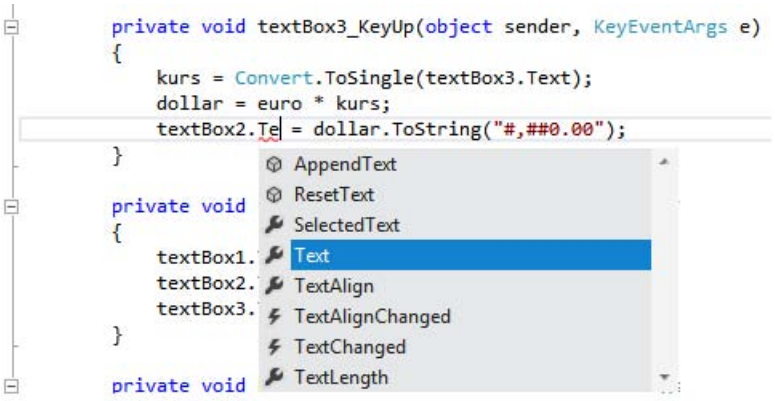
Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen zu verinnerlichen. Da es keine vorgeschriebene Reihenfolge für die Benutzereingaben mehr gibt, werden die anderen Felder sofort aktualisiert. Eine spezielle "="-Schaltfläche (etwa wie bei einem Taschenrechner) ist deshalb überflüssig.

HINWEIS: Achten Sie darauf, dass Sie als Dezimaltrennzeichen das Komma und nicht den Punkt eingeben. Letzterer dient als Tausender-Separator.

IntelliSense – die hilfreiche Fee

Eines der bemerkenswertesten Features des Code-Editors ist seine so genannte *IntelliSense*, auf die Sie mit Sicherheit bereits beim Eintippen des Quellcodes aufmerksam geworden sind. Sobald Sie den Namen eines Objekts bzw. eines Steuerelements mit einem Punkt abschließen, erscheint wie von Geisterhand eine Liste mit allen Eigenschaften und Methoden des Objekts. Das befreit Sie von dem lästigen Nachschlagen in der Hilfe und bewahrt Sie vor Schreibfehlern.

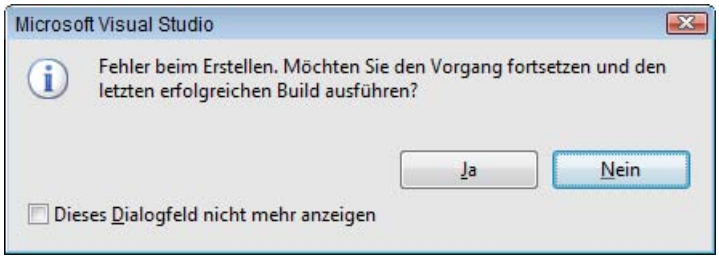
HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.



Hilfe ein Fehler!

Bereits beim Schreiben des Quellcodes werden Sie von Visual Studio auf grundsätzliche Syntaxfehler (z.B. ein vergessenes Semikolon) hingewiesen, im Allgemeinen geschieht dies durch Unterstreichen mit einer wellenförmigen (roten) Linie. Wenn Sie mit der Maus auf diese Linie zeigen, erhalten Sie meistens auch noch einen, mehr oder weniger brauchbaren, Hinweis (z.B. "; wird erwartet").

Andere Fehler treten erst beim Kompilieren ans Tageslicht, wobei Sie durch folgende Meldung aufgeschreckt werden:



In der Regel sollten Sie *Nein* klicken, um unverzüglich den (oder die) Übeltäter im Quellcode zu suchen und dingfest zu machen. Das Lokalisieren ist meist sehr einfach, da die fehlerhaften Ausdrücke durch eine Wellenlinie unterschlängelt sind. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie mit der Maus auf die betreffende Passage zeigen.

HINWEIS: Wenn das Programm sich partout nicht kompilieren lassen will und weit und breit keine Wellenlinie bzw. ein anderer Hinweis auf den Übeltäter in Sicht ist, hilft meist ein Blick in die Fehlerliste am unteren Rand des Hauptfensters (oder Menü *Ansicht|Fehlerliste*).

Auf einen typischen Fehler, der manchen Anfänger zur Verzweiflung bringen kann, soll Sie das folgende Beispiel hinweisen.

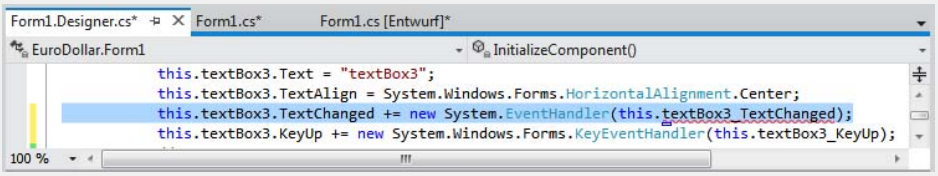
BEISPIEL 1.1: Fehler

Durch einen versehentlichen Doppelklick auf *textBox3* haben Sie im Codefenster unbewusst einen Eventhandler für das *TextChanged*-Ereignis (das ist das Standardereignis für dieses Steuerelement) erzeugt. Sie haben das zwar sofort bemerkt und den Rahmencode des Eventhandlers gleich wieder gelöscht. Trotzdem werden Sie beim anschließenden Versuch, wieder zum Designer umzuschalten, durch folgenden Anblick erschreckt:

Erst ein Blick auf die Fehlerliste bringt Sie auf die richtige Fährte, denn Sie haben zwar den Rahmencode des *textBox3_TextChanged*-Eventhandlers wieder gelöscht, nicht aber seine (von

BEISPIEL 1.1: Fehler

Visual Studio automatisch angelegte) Deklaration. Diese ist nach wie vor vorhanden und verweist (für Sie zunächst unsichtbar) auf den nun nicht mehr vorhandenen Code. Klicken Sie deshalb auf den Link "Gehe zu Code" (oder doppelklicken Sie auf den Eintrag in der Fehlerliste), worauf sich das Codefenster der Datei *Form1.Designer.cs* öffnet[1]:



Der Name des "falschen" Eventhandlers ist mit einer roten Linie unterschlängelt. Entfernen Sie die blau hinterlegte Zeile komplett.

Falls sich anschließend das Designer-Fenster zwar öffnen lässt, auf dem Formular *Form1* aber die Steuerelemente fehlen, so müssen Sie Visual Studio komplett schließen und neu starten.

HINWEIS: Weitere hilfreiche Hinweise zum Debuggen finden Sie im Kapitel 11.

[1] In dieser Datei haben Sie normalerweise nichts zu suchen, da die Einträge von Visual Studio vorgenommen werden.

Kapitel 2

Grundlagen der Sprache C#

In diesem Kapitel wollen wir Ihnen den für den Einstieg wichtigen Sprachkern von C# erklären[1]. Wir zeigen Ihnen, wie Sie Anweisungen schreiben, mit Datentypen umgehen, Schleifen und Verzweigungen einsetzen, Arrays definieren und Funktionen bzw. Prozeduren aufrufen. Mit Rücksicht auf den Einsteiger und um die Übersichtlichkeit nicht zu gefährden, folgen die etwas anspruchsvolleren Bestandteile von C# erst in späteren Kapiteln (objektorientiertes Programmieren in Kapitel 3, fortgeschrittenere Sprachelemente in den Kapiteln 4 und 5).

HINWEIS: Testen Sie möglichst viele der kleinen Codeschnipsel des vorliegenden Kapitels am eigenen PC auf Herz und Nieren. Als Codegerüst kann entweder eine Konsolenanwendung oder aber eine Windows Forms-Anwendung dienen.

2.1 Grundbegriffe

Zur Vorbereitung empfehlen wir ein Rückblättern zum Kapitel 1 (Einführungsbeispiel 1.2, Praxisbeispiele 1.7), wo solch grundlegende Begriffe wie Anweisungen, Klassen, Namensraum und Gültigkeitsbereiche bereits grob erklärt wurden.

2.1.1 Anweisungen

Wir verstehen unter einer Anweisung einen Befehl, der eine bestimmte Aktion ausführt. Wie in jeder anderen Programmiersprache müssen auch die C#-Anweisungen bestimmten Regeln entsprechen, die man in ihrer Gesamtheit als *Syntax*[2] bezeichnet.

Eine der wichtigsten Syntaxregeln kennen Sie bereits aus den Einführungsbeispielen, nämlich dass jede Anweisung mit einem Semikolon abzuschließen ist und dass der Zeilenumbruch dabei keinerlei Rolle spielt.

---

[1] Der Profi, der bereits mit Java, C oder C++ gearbeitet hat, wird dieses Kapitel natürlich mit Siebenmeilenstiefeln durcheilen.

[2] Im Unterschied zur *Syntax* versteht man unter der *Semantik* einer Sprache die Beschreibung dessen, *was* die einzelnen Anweisungen bewirken.

Da C# eine so genannte formatfreie Sprache ist, haben neben dem Zeilenumbruch auch Leerzeichen, Tabulatoren etc. keine Bedeutung, es sei denn, Sie verwenden sie bewusst, um die optische Lesbarkeit Ihres Codes zu verbessern.

Durch gute Strukturierung Ihres Quellcodes, wie z.B. blockweises Einrücken, machen Sie Ihre Programme übersichtlicher und verringern somit die Fehlerquote.

HINWEIS: Die Entwicklungsumgebung von Visual Studio erleichtert Ihnen das blockweise Einrücken unter anderem durch das Menü *Bearbeiten/Erweitert/Zeileneinzug vergrößern* bzw. *verkleinern* oder durch die entsprechenden Schaltflächen der Symbolleiste.

2.1.2 Bezeichner

Für die Namensgebung von Elementen Ihres Programms, wie Variablen, Methoden, Klassen, ... verwenden Sie Bezeichner. Bei der Namensgebung müssen Sie sich an folgende Regeln halten:

- Als Zeichen sind Groß- und Kleinbuchstaben, der Unterstrich "_" sowie die Ziffern 0...9 zulässig.
- Jeder Bezeichner muss mit einem Buchstaben (oder einem Unterstrich) beginnen.
- Als case-sensitive Sprache unterscheidet C# penibel zwischen Groß- und Kleinschreibung.

BEISPIEL 2.1: Zulässige Bezeichner

C#
```
euro
 _radius
zwerg7
```

BEISPIEL 2.2: Unzulässige Bezeichner

C#
```
%Anteil
7Zwerge
Gehalt$
```

Bezüglich der Verwendung von Groß- und Kleinschreibung gibt es zwar keine Verbote, aber folgende Empfehlung:

HINWEIS: Verwenden Sie möglichst keine Bezeichner, die sich lediglich durch die Groß- und Kleinschreibung voneinander unterscheiden!

BEISPIEL 2.3: Beide Bezeichner sollten Sie nicht nebeneinander verwenden:

C#
```
MeineAdresse
meineAdresse
```

2.1.3 Schlüsselwörter

Bei Schlüsselwörtern handelt es sich um vordefinierte reservierte Bezeichner, die den Kern der Sprache C# ausmachen und die im Code-Fenster von Visual Studio normalerweise blau eingefärbt werden. Die folgende (nicht ganz vollständige) Übersicht zeigt die Schlüsselwörter von C#.

abstract	*as*	*async*	*await*	*base*	*bool*
break	*byte*	*case*	*char*	*class*	*const*
continue	*decimal*	*default*	*delegate*	*do*	*double*
else	*enum*	*event*	*explicit*	*extern*	*false*
finally	*fixed*	*float*	*for*	*foreach*	*goto*
if	*implicit*	*in*	*int*	*interface*	*internal*
is	*lock*	*long*	*namespace*	*new*	*null*
object	*operator*	*out*	*override*	*params*	*private*
protected	*public*	*readonly*	*ref*	*return*	*sbyte*
sealed	*short*	*sizeof*	*stackalloc*	*static*	*string*
struct	*switch*	*this*	*throw*	*true*	*try*
typeof	*uint*	*ulong*	*unchecked*	*unsafe*	*ushort*
using	*var*	*virtual*	*void*	*volatile*	*while*

HINWEIS: Schlüsselwörter dürfen Sie **nicht** für selbst definierte Bezeichner verwenden!

Allerdings gibt es zu obigem Hinweis eine gewisse Ausnahme: Wenn Schlüsselwörter ein @ als Präfix enthalten, können sie als Bezeichner im Programm verwendet werden.

BEISPIEL 2.4: Schlüsselwörter mit Präfix

C# *@if* stellt z.B. einen gültigen Bezeichner dar, *if* jedoch nicht, da es sich um ein Schlüsselwort handelt.

2.1.4 Kommentare

Kommentaranweisungen (im Editor standardmäßig grün eingefärbt) dienen als zusätzliche Erläuterungen für den Programmierer, sie sollen die Lesbarkeit des Quellcodes verbessern. Für das Kennzeichnen von Kommentaren können Sie zwei unterschiedliche Verfahren verwenden.

Einzeilige Kommentare

Um eine Zeile (nicht Befehlszeile) als Kommentar zu markieren, leiten Sie diese mit einem doppelten Slash // ein.

BEISPIEL 2.5: Eine Anweisung mit Kommentar

C#
```
private float euro=1, dollar=1, kurs=1;    // Variablendeklaration
```

HINWEIS: Geizen Sie in Ihren Quelltexten nicht mit Kommentaren, damit Sie (oder andere) auch später noch den von Ihnen geschriebenen Code verstehen können!

Mehrzeilige Kommentare

Um einen mehrzeiligen Bereich als Kommentar zu kennzeichnen, wird dieser mit /* und */ eingegrenzt.

BEISPIEL 2.6: Mehrzeiliger Kommentar

C#
```
/* Dieser Kommentar
besteht aus zwei Zeilen */
```

Mehrzeilige Kommentare können Sie auch vorteilhaft beim Testen von Code einsetzen, indem Sie (in der Regel nur vorübergehend) bestimmte Codeabschnitte außer Kraft setzen, d.h. "auskommentieren".

HINWEIS: Die Visual Studio Entwicklungsumgebung erleichtert Ihnen das Auskommentieren von Codeabschnitten mit dem Menü *Bearbeiten/Erweitert/Auswahl kommentieren* (Strg+E, C) bzw. *Auskommentierung der Auswahl aufheben* (Strg+E, U) oder mit den entsprechenden Schaltflächen der Symbolleiste.

2.2 Datentypen, Variablen und Konstanten

Jedes Programm "lebt" in erster Linie von seinen Variablen und Konstanten, die bestimmten Datentypen entsprechen. Es ist daher logisch, dass wir dieses Thema an den Anfang unserer Betrachtungen zur Sprache C# stellen müssen.

2.2.1 Fundamentale Typen

Die folgende Tabelle gibt eine Übersicht der einfachen (fundamentalen) Datentypen, die C# zur Verfügung stellt[1].

Wie Sie der Tabelle entnehmen können, entsprechen alle C#-Datentypen einer Klassendefinition im .NET Framework. Die CLR[2]-Datentypen sind im *System*-Namensraum angeordnet. Bei der Deklaration (siehe unten) ist es egal, welchen der beiden möglichen Typbezeichner Sie angeben.

[1] Auf "anspruchsvollere" Datentypen wie *var* oder *dynamic*, gehen wir erst an späterer Stelle ein.

[2] Die NET-Laufzeitumgebung (*Common Language Runtime*)

C#-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte][1]
byte	*System.Byte*	positive Ganzzahl zwischen 0 ... 255	1
sbyte	*System.SByte*	vorzeichenbehaftete Ganzzahl zwischen -128 ... 127	1
short	*System.Int16*	kurze Ganzzahl zwischen -2^{15} ... $2^{15}-1$ (-32.768 ... 32.767)	2
ushort	*System.UInt16*	vorzeichenlose Ganzzahl zwischen 0 ... 65.535	2
int	*System.Int32*	Ganzzahl zwischen -2^{31} ... $2^{31}-1$ (-2.147.483.648 ... 2.147.483.647)	4
uint	*System.UInt32*	vorzeichenlose Ganzzahl zwischen 0 ... 4.294.967.295	4
ulong	*System.UInt64*	vorzeichenlose Ganzzahl zwischen 0 ... 18.446.744.073.709.551.615	8
long	*System.Int64*	lange Ganzzahl -2^{63} ... $2^{63}-1$ (-9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)	8
float	*System.Single*	einfachgenaue Gleitkommazahl mit 7-stelliger Genauigkeit zwischen ca. +/- 3.4E-45 und +/-3.4E+38	4
double	*System.Double*	doppeltgenaue Gleitkommazahl mit 16-stelliger Genauigkeit zwischen ca. +/- 4.9E-324 und +/- 1.8E+308	8
decimal	*System.Decimal*	hochgenaue Gleitkommazahl zwischen 0 ... +/- 79E+27 (ohne Dezimalpunkt) und ca. +/-1.0E-29 ...7.9E+27 (mit Dezimalpunkt)	16
char	*System.Char*	ein beliebiges Unicode-Zeichen	2
bool	*System.Boolean*	Wahrheitswert (*true*, *false*)	2
string	*System.String*	beliebige Unicode-Zeichenfolge mit einer maximalen Länge von ca. 2.000.000.000 Zeichen	2 pro Zeichen plus 10
object	*System.Object*	universeller Datentyp	4

2.2.2 Wertetypen versus Verweistypen

Mit Ausnahme des *string*- und des *object*-Datentyps, die so genannte *Verweistypen* sind, gehören die übrigen Datentypen in obiger Tabelle zu den *Wertetypen*. Das Verständnis des Unterschieds zwischen diesen beiden fundamentalen Gruppen ist enorm wichtig für das tiefere Eindringen in die Sprache C#.

[1] Systemintern sind auch die ersten vier Typen 4-Byte lang.

Wertetypen

Dazu zählen all die einfachen Datentypen wie *byte*, *int*, *double* ..., hinzu kommen später noch andere wie beispielsweise *struct* (siehe 2.6.2) und *DateTime* (siehe Kapitel 4). Beim Abarbeiten des Programms wird für die lokalen Variablen und die übergebenen Parameter Speicherplatz benötigt, der immer dem Stack entnommen wird. Nach Beendigung einer Methode wird der Speicher automatisch an den Stack[1] zurückgegeben.

Verweistypen

Bislang kennen wir nur die Verweistypen *string* und *object*, im Kapitel 4 kommen später noch Datenfelder (Arrays) hinzu. Aber das ist nur die Spitze des Eisbergs, denn in der objektorientierten Programmierung, in welche wir ab Kapitel 3 tiefer einsteigen werden, sind alle Objekte Verweistypen. Das bedeutet, dass auf dem Stack nicht der Wert des Objekts abgespeichert wird, sondern lediglich ein Verweis (Referenz, Zeiger) auf eine Speicheradresse des Heap, wo das eigentliche Objekt gespeichert ist. Das Anlegen des Objekts auf dem Heap wird auch als Instanziierung bezeichnet, in der Regel muss dazu der *new*-Operator verwendet werden[2]. Das Entsorgen des Speichers übernimmt sporadisch der so genannte Garbage Collector. Doch zu all dem kommen wir erst im nachfolgenden OOP-Kapitel.

2.2.3 Benennung von Variablen

Variablen sind benannte Speicherplatzstellen, der Variablenname dient dazu, die Speicheradresse im Programmcode quasi wie über einen Alias anzusprechen.

Zusätzlich zu den unter 2.1.2 aufgeführten Regeln für selbst definierte Bezeichner sollten Sie folgenden Empfehlungen gemäß *Common Language Specification* (CLS) folgen:

- Beginnen Sie den Namen einer Variablen mit einem Kleinbuchstaben.
- Vermeiden Sie Unterstriche (_).
- Falls Bezeichner aus mehreren Wörtern zusammengesetzt sind, so sollten ab dem zweiten Wort alle Wörter mit einem Großbuchstaben beginnen.

BEISPIEL 2.7: Ein Variablenname, der sich aus mehreren Wörtern zusammensetzt.

```
C# meinHaushaltskassenSaldo
```

2.2.4 Deklaration von Variablen

Variablen werden in C# wie folgt deklariert:

SYNTAX: *Datentyp Variablenname*;

[1] Stack und Heap sind bestimmte Bereiche im Arbeitsspeicher jedes Computers.

[2] Der *string*-Datentyp bildet hier eine gewisse Ausnahme (siehe 4.2).

BEISPIEL 2.8: Drei Variablen unterschiedlichen Datentyps werden deklariert.

C#
```
int Anzahl; double Breite;
string nachName;
```

Wollen Sie mehrere Variablen vom gleichen Datentyp deklarieren, so werden diese durch Kommas separiert.

BEISPIEL 2.9: Drei int-Variablen werden deklariert.

C#
```
int i, j, k;
```

Als Datentyp kann man auch den CLR-Typ angeben (siehe obige Tabelle).

BEISPIEL 2.10: Die folgenden drei Deklarationen sind gleichwertig.

C#
```
int i;
System.Int32 i;
Int32 i;
```

Initialisierte Variablen

Zusammen mit der Deklaration können Sie den Variablen auch gleich Anfangswerte zuweisen (im Fachjargon heißt das "initialisieren").

BEISPIEL 2.11: Initialisierte Variablen

C#

Statt

```
int anzahl;
anzahl = 99;
```

können Sie kürzer formulieren:

```
int anzahl = 99;
```

Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 6 ab Seite 347) neu eingeführte Sprachmerkmal erlaubt es, dass der Datentyp von Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angegeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

HINWEIS: Damit der Compiler den Typ der Variablen feststellen kann, muss eine mit *var* deklarierte Variable unbedingt bei der Deklaration initialisiert werden.

BEISPIEL 2.12: Typinferenz

C#

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden.

BEISPIEL 2.13: Keine Datentypänderung möglich

C#

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *Double*-Wert zugewiesen werden.

```
var b = 7;
b = 12.3;            // Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen, also nicht auf globaler Ebene (siehe Abschnitt 2.2.10).

2.2.5 Typsuffixe

Wenn Sie Variablen im Quellcode direkte Zahlenwerte (Literale) zuweisen wollen, so werden diese vom Compiler standardmäßig als Datentyp *int* bzw. *double* interpretiert. Bei Datentypen wie *long*, *float* oder *decimal* müssen Sie ein so genanntes Typsuffix (*L, F, M*) anhängen, ansonsten werden die Literale wie *int* oder *double* behandelt, und es gibt einen Compilerfehler. Eine Übersicht enthält die folgende Tabelle.

Typsuffix	Gleitkommatyp
f oder *F*	*float*
d oder *D*	*double*
m oder *M*	*decimal*

BEISPIEL 2.14: Richtige und falsche Literalzuweisungen.

C#

```
float max = 99.99;        // Fehler!
float max = 99.99F;       // richtig
decimal geld = 300.50;    // Fehler!
decimal geld = 300.50M;   // richtig
```

2.2.6 Zeichen und Zeichenketten

Die Datentypen *char* und *string* basieren auf dem Unicode-Zeichensatz, der pro Zeichen 2 Byte beansprucht (im Unterschied zum klassischen ASCII- bzw. ANSI-Zeichensatz mit 1 Byte pro Zeichen). Mit dem Unicode können deshalb nicht mehr nur maximal 255, sondern bis zu 65535 (!) verschiedene Zeichen gespeichert werden.

char

Variablen vom *char*-Datentyp können Sie Zeichenliterale, hexadezimale Escape-Sequenzen oder Unicode-Darstellungen zuweisen.

HINWEIS: *char*-Literale sind in Hochkommata (') einzufassen.

BEISPIEL 2.15: char

C#

Drei gleichwertige Anweisungen deklarieren eine *char*-Variable und initialisieren diese mit dem Zeichen *A*:

```
char c = 'A';           // Zeichenliteral
char c = '\x0041';      // hexadezimal
char c = '\u0041';      // Unicode
```

Als weitere Möglichkeit kommt eine explizite Typkonvertierung direkt aus dem (ganzzahligen) Zeichencode in Betracht.

BEISPIEL 2.16: Eine weitere Ergänzung zum Vorgängerbeispiel

C#

```
char c = (char) 65;    // Typcasting liefert 'A'
```

string

HINWEIS: Stringliterale werden in doppelten Hochkommata ("Gänsefüßchen") eingefasst.

BEISPIEL 2.17: Zuweisen einer Stringvariablen

C#

```
string s = "Hallo";
```

Einen einzelnen *char* können Sie direkt aus einem *string* herauskopieren, indem Sie den Index in eckige Klammern schreiben. Dabei hat das erste Zeichen den Index 0.

BEISPIEL 2.18: Das erste Zeichen eines Strings ermitteln

C#

```
string s = "Hallo";
char c = s[0];           // liefert "H"
```

HINWEIS: Ausführliches zur Stringverarbeitung finden Sie in Kapitel 4.

Der umgekehrte Schrägstrich bzw. Backslash (\) spielt innerhalb eines Strings eine besondere Rolle, denn nachfolgende Zeichen werden vom C#-Compiler als Befehl interpretiert[1].

Die folgende Tabelle zeigt die häufigsten in C# benutzten Escapesequenzen.

Escape Sequenz	Bedeutung
\'	einfaches Anführungszeichen
\"	doppeltes Anführungszeichen
\\	Backslash
\a	Signalton
\b	Backspace (BS)
\f	Seitenvorschub
\n	Neue Zeile (LF)
\r	Wagenrücklauf CR)
\t	Horizontaler Tabulator (TAB)

HINWEIS: Bei einem Unicode-Zeichen folgt dem Backslash ein kleines u und die vierstellige Nummer des Zeichens, z.B. '\u0013'.

BEISPIEL 2.19: Korrekte Schreibweise für Dateipfade

C#

Die Anweisung zur Definition eines Dateipfads

```
string path = "C:\Benutzer\Doberenz";
```

... wird bereits von der Entwicklungsumgebung als "nicht erkannte Escapesequenz" zurückgewiesen, da der Backslash als Beginn einer Escapesequenz interpretiert wird.

Die folgende Anweisung wäre korrekt:

```
string path = "C:\\Benutzer\\Doberenz";
```

oder in diesem Fall auch die Kurzform:

```
string path = @"C:\Benutzer\Doberenz";
```

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

C#

```
MessageBox.Show("Beste Grüße von ...\r\n\ahttp://www.doko-buch.de", "Escape Sequenzen",
                MessageBoxButtons.OK, MessageBoxIcon.Information);
```

[1] Siehe dazu auch Kapitel 4, Abschnitt 4 3 (Reguläre Ausdrücke).

BEISPIEL 2.20: Hinzufügen eines Zeilenvorschubs mit Signalton

Ergebnis

Ein Signalton ertönt und folgendes Meldungsfenster erscheint:

2.2.7 object-Datentyp

Der *object*-Datentyp ist weitaus mehr, als es der Name vermuten lässt. Alle Klassen des .NET-Frameworks sind von *System.Object* abgeleitet, und *object* ist dafür lediglich ein Alias.

Eine *object*-Variable ist ein so genannter *Verweis- oder Referenztyp*, denn sie speichert nicht den tatsächlichen Wert, sondern lediglich einen 4 Byte großen Zeiger auf die Adresse der zugewiesenen Variablen.

Variablen des Typs *object* können Sie alles zuweisen.

BEISPIEL 2.21: Einer Objektvariablen *o* wird eine *int*-Zahl zugewiesen.

C#

```
int i = 5;
object o = i;
```

Etwas komplizierter ist der umgekehrte Weg, nämlich wenn Sie den Wert der Objektvariablen einer anderen Variablen zuweisen wollen. In diesem Fall ist man auf explizite Typumwandlung (Typecasting) angewiesen (siehe Abschnitt 2.3.1). Eine direkte Zuweisung (implizite Typkonvertierung) führt zu einem Compilerfehler.

BEISPIEL 2.22: Die Fortsetzung des Vorgängerbeispiels

C#

```
int j = o;          // Fehler!
int j = (int) o;    // mit Typecasting ok!
```

Wichtig in diesem Zusammenhang ist das Verständnis des Unterschieds zwischen Werte- und Referenztypen und dem Prinzip des Boxing/Unboxing (siehe 2.3.6).

2.2.8 Konstanten deklarieren

Im Unterschied zu Variablen bleibt der Wert einer Konstanten während der gesamten Laufzeit eines Programms unverändert. Sie legen ihn einmalig mit dem *const*-Schlüsselwort fest. Die Deklaration ist ähnlich wie bei initialisierten Variablen.

SYNTAX: `const Datentyp Konstantenname = Wert;`

BEISPIEL 2.23: Verschiedene Konstantendeklarationen

C#
```
const int c = 119;
const float PI  = 3.1415F;
const double X1  = 3 * 0.4, X2 = 5.3 + 0.68;
const string s = "Hallo";
```

Sammlungen von Konstanten werden üblicherweise in so genannten *Enumerationen* "zusammengehalten" (siehe Abschnitt 2.6.1).

2.2.9 Nullable Types

C# erfordert seit eh und je die explizite Initialisierung von Variablen.

BEISPIEL 2.24: Falsche und richtige Verwendung von Variablen

C#
```
int z;
z++;              // falsch, weil z nicht initialisiert ist
...
int z = 0;
z++;              // richtig
```

Initialisieren von Wertetypen mit null

Etwas komplizierter wird es aber, wenn man einen Wert mit "nichts" (*null*) initialisieren möchte.

BEISPIEL 2.25: Das funktioniert nicht!

C#
```
int z = null;       // falsch
```

Durch ein der Typdeklaration nachgestelltes Fragezeichen (*?*) kann der Compiler jetzt einen Wertetyp in die generische *System.Nullable*-Struktur verpacken, wodurch es möglich wird, einer Variablen den Wert *null* zuzuweisen.

BEISPIEL 2.26: Aber das funktioniert!

C#
```
int? z = null;
z = 1;
```

Oder als explizite Deklaration:

```
System.Nullable<Int32> z = null;
z = 1;
```

Zuweisungen mit dem ??-Operator

Um einen *Nullable Type* (werteloser Typ) einer anderen Variablen zuweisen zu können, muss vorher eine *null*-Abfrage erfolgen, wie sie mittels *HasValue*-Eigenschaft möglich ist.

BEISPIEL 2.27: Die Variable *y* wird mit der Zahl 0 initialisiert, da *x* den Wert null hat.

C#
```
int? x = null;
int y;
if (x.HasValue) y = (int) x;
else y = 0;
```

Deutlich eleganter und kürzer ist eine solche Zuweisung aber, wenn man dazu das doppelte Fragezeichen (*??*) verwendet, es liefert den Wert des vorangestellten Ausdrucks falls dieser nicht *null* ist, anderenfalls den Wert des nachfolgenden Ausdrucks.

BEISPIEL 2.28: Das Vorgängerbeispiel wird einfacher realisiert.

C#
```
int? x = null;
int y = x ?? 0;
```

BEISPIEL 2.29: Im *Label* erscheint der Text "nichts zugewiesen".

C#
```
string s = null;
label1.Text = s ?? "nichts zugewiesen!";
```

2.2.10 Typinferenz

Dieses in Zusammenhang mit der LINQ-Technologie (siehe Kapitel 6) eingeführte Sprachfeature erlaubt es, dass der Datentyp einer Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass Sie explizit den Typ angeben müssen. Als Ersatz für einen konkreten Typ wird das Schlüsselwort *var* verwendet.

HINWEIS: Damit der Compiler den Typ der Variablen feststellen kann, muss eine mit *var* deklarierte Variable unbedingt bei der Deklaration initialisiert werden.

BEISPIEL 2.30: *var*-Deklaration

C#

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

```
var a = 35;
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
int a = 35;
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden.

BEISPIEL 2.31: Keine Änderung möglich ...

C#

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *double*-Wert zugewiesen werden.

```
var b = 7;
b = 12.3;            // Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen (siehe folgender Abschnitt)!

2.2.11 Gültigkeitsbereiche und Sichtbarkeit

Obwohl sich in C# alles innerhalb von Klassen abspielt, werden wir erst im OOP-Kapitel 3 ausführlicher auf diese Thematik zu sprechen kommen.

Trotzdem sollten Sie bereits jetzt Folgendes wissen:

- Lokale Variablen gelten standardmäßig nur innerhalb ihres – in geschweiften Klammern eingerahmten – Bereichs und der untergeordneten Bereiche. Ein Zugriff von außerhalb ist nicht möglich.
- Sie können die Schlüsselwörter *private* und *public* verwenden um festzulegen, ob auf die Variablen auch von außerhalb zugegriffen werden kann.
- *public*-Konstanten sind nicht empfehlenswert, weil das leicht zu Namenskonflikten führen kann. Wenn Sie auf Klassen- oder Namespace-Ebene mit (globalen) Konstanten arbeiten möchten, verwenden Sie am besten eine Enumeration (siehe Abschnitt 2.6.1).
- Wenn Sie eine Variable nicht als *public* oder *private* deklarieren, ist sie standardmäßig *private*. Man bezeichnet die Schlüsselwörter *private* und *public* auch als *Zugriffsmodifizierer*, sie gelten nicht nur für Variablen, sondern auch für Klassen, Objekte, Eigenschaften und Methoden (siehe Kapitel 3, Abschnitt 3.1.3).
- Im Unterschied zu Visual Basic gibt es in C# keine *Static*-Variablen.

2.3 Konvertieren von Datentypen

C# ist eine typsichere Sprache und nimmt es deshalb mit den Datentypen sehr genau. Schon bei den geringsten Nachlässigkeiten schlagen Ihnen IDE oder Compiler gnadenlos auf die Finger.

2.3.1 Implizite und explizite Konvertierung

Unabhängig vom tatsächlichen Wert, wie er in der Variablen gespeichert ist, lassen sich verschiedene Datentypen nur dann gegenseitig zuweisen, wenn der Wertebereich des rechten Datentyps in den linken "passt". In einem solchen Fall findet eine so genannte *implizite Konvertierung* statt, die der Compiler automatisch vornimmt.

BEISPIEL 2.32: Die Zuweisung *Byte* zu *Integer* funktioniert problemlos.

C#
```
byte b = 100;
int i = b;              // implizite Typkonvertierung
```

Geradezu oberlehrerhaft verhält sich C# im umgekehrten Fall. Egal ob der Wert in den kleineren Datentyp passt oder nicht – es wird halt gemeckert.

BEISPIEL 2.33: Das geht nicht

C#

Obwohl der Wert 100 problemlos in eine *Byte*-Variable passt, erscheint die Fehlermeldung "Implizite Konvertierung des Typs 'int' zu 'byte' nicht möglich!"

```
int i = 100;
byte b = i;             // Fehler!
```

Um den meckernden Compiler zu beschwichtigen, ist eine so genannte *explizite Typkonvertierung* (auch *Typecasting* genannt) erforderlich.

SYNTAX: *neueVariable* = (*Neuer Datentyp*) *alteVariable*;

BEISPIEL 2.34: Das Vorgängerbeispiel wird fehlerfrei ausgeführt

C#
```
int i = 100;
byte b = (byte) i;      // explizite Typkonvertierung
```

Implizite Konvertierungen sind sicher, Datenverluste sind deshalb ausgeschlossen. Dabei kann stets nur der kleinere der beiden Datentypen direkt in einen größeren umgewandelt werden[1].

Explizite Typkonvertierungen sollten stets mit Vorsicht angewendet werden, wobei man sicher sein muss, dass die Wertebereiche zur Laufzeit nicht überschritten werden.

HINWEIS: Man muss sich immer bewusst sein, dass eine explizite Typkonvertierung dann zu Datenverlusten führen kann, wenn der Wertebereich durch die Konvertierung verkleinert wird.

BEISPIEL 2.35: Da der *byte*-Datentyp nur den Bereich 0 ... 255 abdeckt, entsteht ein falsches Ergebnis

C#
```
int i = 300;
byte b = (byte) i;      // liefert falsches Resultat (44)
```

[1] Sie können sich das bildlich so vorstellen, dass jeder Datentyp einem Kochtopf mit unterschiedlichem Fassungsvermögen entspricht, und Sie dürfen immer nur etwas aus einem kleineren in einen größeren Topf füllen. Verboten wäre es beispielsweise, aus einem 1-Liter-Topf etwas in einen 0,5-Liter-Topf zu gießen, obwohl im 1-Liter-Topf nur 0,1 Liter enthalten sind!

BEISPIEL 2.36: Implizite und explizite Typkonvertierung *float* in *int* gegenübergestellt

```
C#
int i;
float f = 12.5F;
i = f;              // implizite Konvertierung ergibt Fehler!
i = (int) f;        // explizite Konvertierung ergibt Datenverlust: i erhält den Wert 12
```

BEISPIEL 2.37: Implizite Typkonvertierung *char* in *int*

```
C#
char c = 'A';
int i = c;                  // i erhält den Wert 65  (Zeichencode von 'A')
```

BEISPIEL 2.38: Explizite Typkonvertierung des Ergebnisses einer Division

```
C#
int i = 3;
float x = i / 10;             // x erhält den Wert 0
float x = (float) i / 10;     // x erhält den Wert 0,3
```

as-Konverierungsoperator

Eine Alternative zur expliziten Typumwandlung mittels *()*-Konvertierung ist der *as*-Operator, der allerdings nur auf Verweis- und nicht auf Wertetypen anwendbar ist. Auch alle Steuerelemente gehören zu den Verweistypen!

BEISPIEL 2.39: Konvertieren des *sender*-Parameters eines Eventhandlers

```
C#
this.Text = (sender as TextBox).Text;
```

2.3.2 Welcher Datentyp passt zu welchem?

Der folgenden Tabelle entnehmen Sie alle möglichen impliziten und expliziten Typkonvertierungen. Die impliziten Konvertierungen sind fett hervorgehoben.

Quell-Datentyp	Zieldatentypen
bool	*object*
byte	*ushort, short, uint, int, ulong, long, float, double, decimal, object, sbyte, char*
sbyte	***short, int, long, float, double, decimal, object,*** *byte, ushort, uint, ulong, char*
short	***int, long, float, double, decimal, object,*** *sbyte, byte, ushort, uint, ulong, char*
ushort	***uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short, char*
char	***ushort, uint, int, ulong, long, float, double, decimal, object,*** *sbyte, byte, short*
int	***long, float, double, decimal, object,*** *sbyte, byte, short, ushort, uint, ulong, char*
uint	***long, float, double, decimal, object,*** *sbyte, byte, short, ushort, int, char*

Quell-Datentyp	Zieldatentypen
long	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, ulong, char*
ulong	***float, double, decimal, object,*** *sbyte, byte, short, ushort, int, uint, long, char*
float	***double, object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, decimal*
double	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, decimal*
decimal	***object,*** *sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double*
string	*object*
object	alle Datentypen (Unboxing, siehe Abschnitt 2.3.6)

2.3.3 Konvertieren von string

Laut obiger Konvertierungstabelle lässt sich der *string*-Datentyp nur in den universellen *object*-Datentyp umwandeln und in umgekehrter Richtung scheint gar nichts zu gehen Doch die Entwarnung folgt zugleich.

ToString-Methode

Der *object*-Datentyp – gewissermaßen die "Mutter aller Objekte" – vererbt an alle Nachkommen die *ToString*-Methode, auf welche Sie bereits hin und wieder in den bisherigen Beispielen gestoßen sind, nämlich dann, wenn es darum ging, Zahlenwerte zur Anzeige zu bringen.

HINWEIS: Jeder Datentyp kann mittels seiner *ToString*-Methode in den Datentyp *string* umgewandelt werden!

Und noch ein Hinweis, den sich besonders die von Visual Basic kommenden Umsteiger hinter die Ohren schreiben sollten:

HINWEIS: Vergessen Sie nicht die Klammern hinter *ToString()*!

BEISPIEL 2.40: Anzeige einer Gleitkommazahl

C#
```
double d = 12.75;
MessageBox.Show(d.ToString);     // Fehler
MessageBox.Show(d.ToString());   // ok
```

BEISPIEL 2.41: Konvertieren eines *bool* in *string*

C#
```
bool b = true;
string s = b.ToString();         // "True"
```

String in Zahl verwandeln

Zwar können wir mit der *ToString()*-Methode alle Datentypen in den *string*-Typ konvertieren, wie aber sieht es umgekehrt aus?

Für bestimmte andere Datentypen gibt es spezifische Lösungen, z.B. zum Umwandeln von *string* in *char*.

BEISPIEL 2.42: Einem *char* wird das zweite Zeichen eines *string* zugewiesen.

C#
```
string name = "Max";
char c = name[1];
MessageBox.Show(c.ToString());      // zeigt "a"
```

Damit enden vorerst unsere Erfolgserlebnisse, denn das übliche Typecasting scheint bei den anderen Datentypen zu versagen.

BEISPIEL 2.43: Das geht leider nicht.

C#
```
string s = "5";
int i = (int) s;           // Fehler!
```

Rettung naht auch hier in Gestalt der *Convert*-Klasse (siehe auch nächster Abschnitt). Als Alternative zu den expliziten Typkonvertierungen bietet diese Klasse für nahezu jeden Datentyp eine spezielle (statische) Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

BEISPIEL 2.44: Das Vorgängerbeispiel kann wie folgt gelöst werden.

C#
```
string s = "5";
int i = Convert.ToInt32(s);
MessageBox.Show(i.ToString());      // zeigt "5"
```

BEISPIEL 2.45: Ein *string* wird in eine *double*-Zahl konvertiert.

C#
```
string s = "23,50";
double d = Convert.ToDouble(s);     // d erhält den Wert 23,50
```

Alternativ kann auch die *Parse*-Methode eingesetzt werden.

BEISPIEL 2.46: Konvertieren eines Stringliterals in eine Ganzzahl.

C#
```
int nr = Int32.Parse("12");
```

EVA-Prinzip

Auch für (fast) jedes Programm gilt nach wie vor das uralte EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe). In diesem Zusammenhang sei nochmals auf die besondere Bedeutung der Typkonvertierung von und in den *string*-Datentyp hingewiesen. Da unter Windows sehr häufig die Übergabe-

werte als Zeichenketten vorliegen (*Text*-Eigenschaft der Ein- und Ausgabefelder), müssen sie zunächst in Zahlentypen umgewandelt werden, um dann nach ihrer Verarbeitung wieder in Zeichenketten rückverwandelt und (formatiert) zur Anzeige gebracht zu werden.

BEISPIEL 2.47: Ein Ausschnitt aus dem Einführungsbeispiel 1.7.2

C#
```
euro = Convert.ToSingle(textBox1.Text);         // Eingabe: string => float
dollar = euro * kurs;                           // Verarbeitung
textBox2.Text = dollar.ToString("#,##0.00");    // Ausgabe: float => string
```

2.3.4 Die Convert-Klasse

Diese statische Klasse bietet für jeden einfachen Datentyp eine spezielle Konvertierungsmethode, der man den zu konvertierenden Ausdruck als Argument übergibt.

SYNTAX: `Convert.`*`typMethode`*`(object `*`expr`*`);`

typeMethode = Konvertierungsmethode (*ToBoolean*, *ToByte, ToInt32, ToDouble* ...)

expr = zu konvertierender Ausdruck

BEISPIEL 2.48: *string* wird in *double* konvertiert

C#
```
string s = "55,7";
double d = Convert.ToDouble(s);          // 55,7
```

BEISPIEL 2.49: *bool* wird in *int* und in *string* konvertiert

C#
```
bool b = true;
int i = Convert.ToInt32(b);             // 1
b = false;
i = Convert.ToInt32(b);                 // 0
string s = Convert.ToString(b);         // "False"
```

2.3.5 Die Parse-Methode

Die numerischen Typen *Byte*(*byte*), *Int16*(*short)*, *Int32(int)*, *Int64*(*long)*, *Single*(*float)* und *Double*(*double*) verfügen u.a. über die (statische) *Parse*-Methode, welche die Stringdarstellung einer Zahl in den entsprechenden Typ konvertieren kann.

BEISPIEL 2.50: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

C#
```
double z = Double.Parse(textBox1.Text);
```

HINWEIS: Die *Parse*-Methode hat den Vorteil, dass zusätzlich Kulturinformationen eines bestimmten Landes mit übergeben werden dürfen.

2.3.6 Boxing und Unboxing

Die Begriffe *Boxing/Unboxing* gehören zu den häufig strapazierten .NET-Schlagwörtern, die manchem Einsteiger Ehrfurcht einflößen. Was verbirgt sich dahinter? Sie wissen bis jetzt, dass Sie dem universellen *object*-Datentyp jeden Wert direkt zuweisen können, d h. durch implizite Typkonvertierung. Umgekehrt kann, falls es der *object*-Inhalt erlaubt, jeder Datentyp durch explizite Typkonvertierung (Typecasting) aus *object* wieder "herausgezogen" werden. Das direkte Zuweisen funktioniert in diesem Fall nicht.

BEISPIEL 2.51: Boxing und Unboxing

C#

Eine *bool*-Variable wird in ein *object* "verpackt" (Boxing) und dieses anschließend einer zweiten *bool*-Variablen zugewiesen (Unboxing).

```
bool b1 = true;
object o = b1;          // ok, implizite Konvertierung (Boxing)
bool b2 = o;            // Fehler, implizite Konvertierung
bool b2 = (bool) o;     // ok, explizite Konvertierung (Unboxing, b2 ist true)
```

Um den tieferen Sinn von Boxing/Unboxing zu verstehen, sollten Sie sich nochmals den Unterschied zwischen den beiden fundamentalen Arten von Datentypen vergegenwärtigen, d h., zwischen den Wertetypen und den Verweis- bzw. Referenztypen (siehe Abschnitt 2.2.2).

Boxing

Es erhebt sich nun die Frage, was denn passiert, wenn man einer *object*-Variablen, d.h. einem Verweistyp, einen Wertetyp zuweist, der naturgemäß im Stack gespeichert ist.

BEISPIEL 2.52: Ein Integer wird einem object-Datentyp zugewiesen.

C#

```
int i = 25;
object o = i;
```

Die genauere Fragestellung ist, worauf zeigt die *object*-Variable o? Der Zeiger *o* darf doch keinesfalls auf den Stack verweisen (das würde die Stabilität des Programms massiv gefährden)!

Die Antwort: Es findet ein automatischer Kopiervorgang statt, d h., eine Kopie der Variablen *i* wird auf dem Heap abgelegt, auf die dann die *object*-Variable *o* zeigt.

Unboxing

Wie greift man nun aber wieder auf den in der *object*-Variablen "eingepackten" Wert zu? Eine einfache (implizite) Zuweisung funktioniert nicht. Richtig ist eine explizite Typkonvertierung (Typecasting).

BEISPIEL 2.53: Das Vorgängerbeispiel wird fortgesetzt.

C#

```
int j = o;                  // Fehler!
int j = (int)  o;           // ok
```

Allerdings funktioniert das Typecasting nur dann, wenn die Objektvariable tatsächlich auf den gewünschten Typ verweist, Trickserei – wie im folgenden Beispiel – nützt Ihnen also nichts.

BEISPIEL 2.54: Das geht nicht!

C#

Die Hoffnung, bei der Umwandlung *string* nach *int* vielleicht ohne *Convert*-Klasse (siehe oben) auszukommen, geht leider nicht in Erfüllung.

```
string s = "5";
object o = s;
int i = (int) o;    // Fehler!
```

HINWEIS: Das Boxing ist mit ein wesentlicher Grund, warum in .NET "alles ein Objekt" ist, denn auch Wertetypen können damit quasi wie Objekte behandelt werden.

BEISPIEL 2.55: Ja, auch das funktioniert!

C#

```
int i = new int();
i = 12;
```

2.4 Operatoren

Operatoren verknüpfen Variablen bzw. Operanden miteinander und führen Berechnungen durch. Wir unterscheiden zwischen

- arithmetischen Operatoren,
- Zuweisungsoperatoren,
- logischen Operatoren und
- Vergleichsoperatoren.

Die meisten Operatoren in C# benötigen zwei Operanden.

BEISPIEL 2.56: Operanden

C#

Im Ausdruck

```
i = 12;
```

ist der *Operator* das Gleichheitszeichen (=), die beiden *Operanden* sind die Variable *i* und die Literalkonstante *12*.

HINWEIS: C# erlaubt auch das Überladen von Operatoren, auf das wir aber erst an späterer Stelle eingehen wollen (siehe Kapitel 5).

2.4.1 Arithmetische Operatoren

Standard-Operatoren

Es gibt zunächst die üblichen Operatoren für die Grundrechenarten:

Operator	Beispielausdruck	Erklärung
+	*x + y*	Addition
-	*x - y*	Subtraktion
*	*x * y*	Multiplikation
/	*x / y*	Division
%	*x % y*	Modulo-Division (liefert Restwert)

BEISPIEL 2.57: Standard-Operatoren

```
C#
int i, j = 6;
i = 3 *(4 + 5) * j;    // 162
i = 7 % 3;             // 1 (Rest!)
```

HINWEIS: Achten Sie bei der Division von Literalen darauf, dass das Ergebnis abgerundet wird, wenn nicht mindestens einer der Operanden als Gleitkommatyp gekennzeichnet ist.

BEISPIEL 2.58: Nur die beiden letzten Divisionen liefern das exakte Ergebnis.

```
C#
double d;
d = 7 / 3;             // 2   (Ergebnis wird abgerundet!)
d = 7D / 3;            // 2,3333333333333
d = 7.0 / 3;           //       dto.
```

Inkrement- und Dekrement-Operatoren

Mit den Kurz-Operatoren ++ und -- lässt sich das schrittweise Erhöhen (Inkrementieren) bzw. Erniedrigen von Variablen (Dekrementieren) vereinfachen.

Operator	Beispielausdruck	Erklärung
++	x++ ++x	Postfix-Inkrement Präfix-Inkrement
--	x-- --x	Postfix-Dekrement Präfix-Dekrement

Wie Sie den Beispielen in obiger Tabelle entnehmen, können Sie die Kurz-Operatoren ++ und -- nicht nur hinter den Namen der Variablen (Postfix), sondern auch davor (Präfix) schreiben.

BEISPIEL 2.59: Postfix-Inkrement und Postfix-Dekrement

C#
```
int i = 10;
i++;                // i erhält den Wert 11
double d = 2.5;
d--;                //  d erhält den Wert 1,5
```

BEISPIEL 2.60: Äquivalente Version des Vorgängerbeispiels mit Präfixoperationen

C#
```
int i = 10;
++i;                // i erhält den Wert 11
double d = 2.5;
--d;                //  d erhält den Wert 1,5
```

Wie Sie sehen, haben beide Schreibweisen keinerlei Einfluss auf den Wert der Variablen, diese wird in jedem Fall um 1 inkrementiert bzw. dekrementiert, wozu also sollen Postfix- und Präfix-Notationen dann gut sein?

Um den "feinen" Unterschied zu verstehen, muss man wissen, dass nicht nur die Variable (z.B. *i*) einem bestimmten Wert entspricht, sondern auch die mit dem Kurz-Operator verknüpfte Variable (z.B. *i*++). Letztere hat bei einer Postfix-Operation den Wert **vor** der Inkrementierung bzw. Dekrementierung, bei einer Präfix-Operation hingegen den Wert **nach** Inkrementierung bzw. Dekrementierung.

BEISPIEL 2.61: Ergebnisse einer Postfix-Inkrementation werden im Meldungsfenster angezeigt.

C#
```
int i = 10;
MessageBox.Show((i++).ToString());   // i++ hat den Wert 10
MessageBox.Show(i.ToString());       // i hat den Wert 11
```

BEISPIEL 2.62: Dasselbe für eine Präfix-Inkrementation.

C#
```
int i = 10;
MessageBox.Show((++i).ToString());     // ++i hat den Wert 11
MessageBox.Show(i.ToString());         // i hat den Wert 11
```

Eine besondere Rolle spielen Postfix- und Präfix-Schreibweise bei der Steuerung von *while*- und *do*-Schleifen (siehe Abschnitt 2.5.2).

2.4.2 Zuweisungsoperatoren

Die Tabelle zeigt, dass es neben dem simplen Zuweisungsoperator (=) noch fünf andere mit arithmetischen Operatoren verknüpfte Kurz-Operatoren gibt.

Operator	Beispielausdruck	Erklärung
=	x = y	x wird der Wert von y zugewiesen
+=	x += y	x ergibt sich zu x + y

Operator	Beispielausdruck	Erklärung
-=	x -= y	x ergibt sich zu x - y
*=	x *= y	x ergibt sich zu x * y
/=	x /= y	x ergibt sich zu x / y
%=	x %= y	x ergibt sich als Restbetrag aus x/y

Die Kurz-Operatoren bringen relativ bescheidene Verbesserungen, sie erleichtern das Schreiben von Quellcode und erhöhen die Übersichtlichkeit.

BEISPIEL 2.63: Kurz-Operatoren

C#

Statt

```
i = i + 3;
```

kann man auch schreiben

```
i += 3;
```

BEISPIEL 2.64: Kurz-Operatoren

C#

Für

```
string s = "Hallo";
```

gibt es diese

```
s = s + " .NET-Freunde!";
```

oder diese Möglichkeit zum Anhängen einer weiteren Zeichenkette:

```
string s = "Hallo";
s += " .NET - Freunde!";              // "Hallo .NET - Freunde!"
```

2.4.3 Logische Operatoren

Logische Operatoren basieren auf Ja-/Nein- bzw. *true-/false*-Werten. In C# ist dazu ein reichhaltiges Angebot enthalten.

Vergleichsoperatoren

Vergleichs- oder relationale Operatoren vergleichen zwei Ausdrücke miteinander und liefern als Ergebnis einen *true/false*-Wahrheitswert.

Operator	Erklärung
==	x gleich y?
!=	x ungleich y?
<	x kleiner y?

Operator	Erklärung
<=	x kleiner oder gleich y?
>	x größer y?
>=	x größer oder gleich y?

BEISPIEL 2.65: Vergleich von zwei Integer-Zahlen

C#
```
bool b = 10 < 5;                // b ist false
```

Besondere Bedeutung haben Vergleichsoperatoren im Zusammenhang mit Verzweigungsbefehlen, wie wir sie in Abschnitt 2.5.1 kennen lernen werden. Das folgende Beispiel liefert einen Vorgeschmack.

BEISPIEL 2.66: Vergleichsoperatoren im Zusammenhang mit Verzweigungsbefehlen,

C#

Wenn der Wert der Variablen min gleich 59 ist, wird ihr Wert auf 0 gesetzt, ansonsten um 1 erhöht.

```
if (min == 59) min = 0; else min++;
```

Obwohl *string* ein Verweistyp ist, werden die Gleichheitsoperatoren (== und !=) so definiert, dass die Werte von *string*-Objekten und keine Verweise verglichen werden.

BEISPIEL 2.67: Zwei Strings werden verglichen.

C#
```
string a = "Hallo";
string b = "H";
b += "allo";                          // b wird zu "Hallo"
Console.WriteLine( a == b );          // liefert true, da die Werte gleich sind
Console.WriteLine( (object) a == b ); // liefert false, da es verschiedene Objekte sind
```

Boolesche Operatoren

Diese Operatoren werden auf boolesche Variablen (*true/false*) angewendet:

Operator	Erklärung
&	**Und:** liefert *true*, wenn beide Operanden *true* sind
|	**Oder:** liefert *true*, wenn mindestens einer der Operanden *true* ist
^	**Exklusiv-Oder (XOR):** liefert *true*, wenn genau nur einer der beiden Operanden *true* ist
&&	**intelligentes Und:** wie &-Operator, aber ist der erste Operand *false*, wird der zweite nicht ausgewertet
||	**intelligentes Oder:** wie |-Operator, aber ist der erste Operand *true*, wird der zweite nicht ausgewertet
!	**Negation:** aus *true* wird *false* und aus *false* wird *true*.

BEISPIEL 2.68: Boolesche Operatoren

C#
```
bool b = (true & true) | (false & true);      // b wird true
bool z = (10 < 5) ^ (11 > 11)                 // z wird false
bool schalter = true;
schalter = !schalter;                         // schalter wird false
```

Kurzschlussauswertung

Ist bei einer UND-Verknüpfung der linke Teil *false*, so kann auf die Auswertung des rechten Teils verzichtet werden, da das Ergebnis sowieso *false* ist. Ist bei einer ODER-Verknüpfung der linke Teil *true*, so steht ebenfalls das Ergebnis bereits fest (*true*).

Diese Gesetzmäßigkeit machen sich die "intelligenten" Verknüpfungsoperatoren && und || zunutze, indem sie ein auch als *Kurzschlussauswertung* bekanntes Verfahren verwenden. Wenn der linke Teil bereits zu einem eindeutigen Ergebnis führt, wird der rechte Teil gar nicht erst ausgewertet.

BEISPIEL 2.69: Kurzschlussauswertung

C#

Es wird das gleiche Ergebnis wie im Vorgängerbeispiel erzielt, aber es wird weniger Rechenzeit benötigt, da der rechte Teil nicht ausgewertet wird (der linke Teil ist *true*).

```
bool b = (true && true) || (false && true);         // b wird true
```

HINWEIS: Verwenden Sie die Operatoren && und || anstatt & und |, da Sie dadurch Rechenzeit einsparen!

Bitweise Operationen

Mit den folgenden Operatoren (von denen Ihnen die ersten drei bereits bekannt sind) lassen sich bitweise Verknüpfungen durchführen. Sie verknüpfen also nicht mehr die booleschen Variablen *true* und *false*, sondern die einzelnen Bits (0 bzw. 1) von zwei Zahlen.

Operator	Erklärung
&	bitweise "UND"-Verknüpfung der beiden Operanden
|	bitweise "ODER"-Verknüpfung der beiden Operanden
^	bitweise "XOR"-Verknüpfung der beiden Operanden
>>	Rechtsverschiebung aller Bits eines Operanden um eine bestimmte Anzahl
<<	Linksverschiebung aller Bits eines Operanden um eine bestimmte Anzahl

BEISPIEL 2.70: Die XOR-Verknüpfung der Integer-Zahlen 1 und 7 ergibt 6.

C#
```
int a, b;
a = 1;                  // Bitmuster = 001
b = 7;                  // Bitmuster = 111
```

BEISPIEL 2.70: Die XOR-Verknüpfung der Integer-Zahlen 1 und 7 ergibt 6.

```
a = a ^ b;              // Bitmuster = 110  (a erhält den Wert 6)
```

BEISPIEL 2.71: Die Bitfolge der Zahl 1 wird um zwei Stellen nach links "geshiftet" und ergibt 4.

C#
```
int a = 1;             // Bitmuster = 001
a = a << 2;            // Bitmuster = 100    (a erhält den Wert 4)
```

2.4.4 Rangfolge der Operatoren

Es ist klar, dass bei einem Zuweisungsoperator (=) immer erst die rechte Seite ausgerechnet und dann der linken Seite zugewiesen wird. Aber in welcher Reihenfolge werden die Operationen auf der rechten Seite ausgeführt? Antwort gibt die folgende Tabelle, welche die Operatoren in ihrer hierarchischen Rangfolge zeigt.

Operator	Bedeutung
( )	Klammern
!	logisches NOT
* / %	Multiplikation, Division, Modulo
+ -	Addition, Subtraktion
< <= > >=	kleiner als, kleiner gleich als, größer als, größer gleich als
== !=	gleich, ungleich

Die weiter oben in der Hierarchie stehenden Operationen werden immer **vor** den weiter unten stehenden ausgeführt.

HINWEIS: Durch Einschließen in runde Klammern ( ) kann die hierarchische Reihenfolge außer Kraft gesetzt werden.

BEISPIEL 2.72: Arithmetische Operationen

C#
```
double x = 2.0;
double y  = x * x + 1 + x / 4;              // y = 5,5
```

aber

```
double y  = x * (x + 1 + x / 4);            // y = 7
```

BEISPIEL 2.73: Boolesche Operationen

C#
```
bool b = !true && false || 5 > 6;           // b = false
int z = 50;
bool numeric = z > 47 && z < 58;            // numeric = true
```

2.5 Kontrollstrukturen

Verzweigungs- und Schleifenanweisungen unterbrechen den linearen Programmablauf und gehören zum Einmaleins des Programmierens.

2.5.1 Verzweigungsbefehle

"Programmweichen" werden durch Verzweigungsanweisungen bzw. -funktionen oder auch durch Sprungbefehle gestellt.

Klassische Entscheidungsanweisungen

Die folgende Tabelle gibt einen Überblick[1].

Verzweigungsanweisung	Erklärung

`if (Bedingung)`
`  Anweisung1;`
`else`
`  Anweisung2;`
	bedingte Verzweigung
	Wenn die *Bedingung* zutrifft, wird *Anweisung1* ausgeführt, ansonsten *Anweisung2*.
	Der *else*-Zweig kann auch weggelassen werden.

`if (Bedingung1)`
`  Anweisung1;`
`else if (Bedingung2)`
`  Anweisung2;`
`else if (Bedingung3)`
`  Anweisungen ...;`
	verschachtelte Verzweigung
	Zweige werden so lange ausgewertet, bis eine der Bedingungen *true* ergibt

`switch (Ausdruck)`
`{`
`  case Ausdruck1 :`
`  Anweisungen;`
`  break;`
`  case Ausdruck2 :`
`  Anweisungen;`
`  break;`
`  ...`
`  default :`
`  Anweisungen;`
`  break;`
`}`
	Blockstruktur
	Der Ausdruck wird mit den hinter *case* angeführten Ausdrücken verglichen. Nach erstem Erfolg wird der Block verlassen, ansonsten werden die *default*-Anweisungen ausgeführt.
	Der *default*-Zweig kann auch weggelassen werden, in diesem Fall erfolgt eine Fortsetzung nach Blockende.
	Die *break*-Befehle verhindern das "Durchfallen".

In vielen Fällen werden Sie zum Prüfen von Bedingungen die *if*-Anweisung verwenden.

[1] Statt der einzelnen Anweisungen können auch Anweisungsblöcke stehen, die dann in geschweifte Klammern einzuschließen sind.

BEISPIEL 2.74: In den Labels wird "Verbessern" und "Du bekommst nichts!" angezeigt

C#
```
int zensur = 3;
if (zensur == 1)
{
    label1.Text = "Gratuliere!";
    label2.Text = "Du bekommst einen Blumenstrauß!";
}
else
{
    label1.Text = "Verbessern!" ;
    label2.Text = "Du bekommst nichts!";
}
```

Optional können Sie innerhalb des *if*-Blocks noch *else*- oder *else if*-Zweige verwenden, wobei die *else if*-Bedingung nur dann geprüft wird, wenn keine der vorstehenden *if*-Bedingungen erfüllt war.

BEISPIEL 2.75: Im Label wird "Befriedigend" angezeigt

C#
```
int zensur = 3;
if (zensur == 1)
    label1.Text =  "Sehr gut!";
else if (zensur == 2)
    label1.Text =  "Gut";
else if (zensur == 3)                          // zutreffende Bedingung
    label1.Text =  "Befriedigend";
    //(usw.)
else
  label1.Text = "Nicht erlaubte Zensur!";
```

Mit dem *switch*-Konstrukt wird ein Ausdruck auf mehrere mögliche Ergebnisse hin überprüft. Im Testausdruck kann ein beliebiger arithmetischer oder logischer Ausdruck stehen.

BEISPIEL 2.76: Diese Kontrollstruktur leistet das Gleiche wie das Vorgängerbeispiel

C#
```
int zensur = 3;
switch (zensur) {
    case 1: label1.Text = "Sehr gut"; break;
    case 2: label1.Text =  "Gut"; break;
    case 3: label1.Text =  "Befriedigend"; break;     // zutreffende Bedingung
    //(usw.)
    default : label1.Text = "Nicht erlaubte Zensur!"; break;
}
```

HINWEIS: Sie können *switch* nur bei einfachen Datentypen wie *byte* und *int* sowie *string* verwenden. In allen anderen Fällen müssen Sie *if*-Konstrukte nehmen.

Um eine identische Aktion bei mehreren möglichen Vergleichswerten auszuführen, schreiben Sie die einzelnen *case*-Zweige einfach hintereinander und lassen dabei das Schlüsselwort *break* weg.

BEISPIEL 2.77: Das *Label* zeigt "Frühling" an

C#
```
byte monat = 5;
switch (monat) {
    case 12 : case 1 :
    case 2 : label1.Text =  "Winter"; break;
    case 3 : case 4 :
    case 5 : label1.Text =  "Frühling"; break;       // zutreffende Bedingung
    case 6 : case 7 :
    case 8 : label1.Text =  "Sommer"; break;
    case 9 : case 10 :
    case 11: label1.Text =  "Herbst"; break;
    default :
    label1.Text =  "kein gültiger Monat!"; break;
}
```

HINWEIS: Sie sollten, wo immer es geht, statt einer *if*-Anweisung mit eingeschachtelten *else if*-Verzweigungen eine *switch*-Anweisung verwenden. Diese wird wesentlich schneller ausgeführt, da die Prüfbedingung nur einmal auszuwerten ist.

In der Prüfbedingung für das *if*-Konstrukt wird auch oft vom Negations-Operator (*!*) Gebrauch gemacht:

BEISPIEL 2.78: An den Verzeichnisnamen *myPath* wird ein Slash (/) angehängt, falls keiner vorhanden ist

C#
An den Verzeichnisnamen *myPath* wird ein Slash (/) angehängt, falls keiner vorhanden ist.
```
if (!myPath.EndsWith("/")) myPath += "/";
```

Ergänzung

Ein weniger gebräuchlicher Verzweigungsbefehl basiert auf dem Fragezeichen (?) und durch Doppelpunkt (:) getrennten Zielanweisungen. Dadurch lässt sich eine kompaktere Schreibweise erzwingen.

BEISPIEL 2.79: Der Verzweigungsbefehl

C#
```
label1.Text = checkBox1.Checked ? "Ja"  : "Nein";
```
entspricht
```
if (checkBox1.Checked) label1.Text = "Ja";
else label1.Text = "Nein";
```

BEISPIEL 2.80: In Abhängigkeit von einer booleschen Variablen erhält i den Wert 1 oder 2

C#
```
int i = checkBox1.Checked ? 1 : 2;
```

Sprungbefehle

Verzweigungen können auch mit Sprunganweisungen realisiert werden. Innerhalb von Sprunganweisungen werden die Schlüsselwörter *continue*, *default*, *goto* und *return* eingesetzt.

So sind innerhalb eines *switch*-Konstrukts auch absolute Sprünge mittels *goto* möglich. Die *case*- oder *default*-Anweisungen sind die Sprungziele.

BEISPIEL 2.81: Eine Alternative zum Vorgängerbeispiel (auszugsweise)

C#
```
switch (monat)
{
  case 12: goto case 2;
  case 1:  goto case 2;
  case 2 : label1.Text =  "Winter"; break;
  case 3 : goto case 5;
  case 4 : goto case 5;
  case 5 : label1.Text =  "Frühling"; break;
// usw.
```

2.5.2 Schleifenanweisungen

Die wichtigsten Grundtypen sind *for*-, *while*- und *do*-Schleifen (siehe Tabelle).

Schleifenanweisung	Erklärung

`for (Zähler=Anfangswert; Abbruchbedingung; Zähler=neuerWert)`
`{`
`  Anweisungen;`
`}`
	***for*-Zählschleife,**
	wird so lange durchlaufen, bis Abbruchbedingung *false* ist

`while (Abbruchbedingung)`
`{`
`  Anweisungen;`
`}`
	***while*- Bedingungsschleife,**
	Abbruchbedingung am Schleifenanfang (kopfgesteuert)

`do`
`{`
`  Anweisungen;`
`}`
`while (Abbruchbedingung)`
	***do*- Bedingungsschleife,**
	Abbruchbedingung am Schleifenende (fußgesteuert)

HINWEIS: Ein weiterer Schleifentyp, die *foreach*-Schleife, spielt im Zusammenhang mit Arrays und Auflistungen eine wichtige Rolle (siehe Kapitel 4).

for-Schleifen

In diesem klassischen Schleifentyp wird die Zählervariable pro Durchlauf aktualisiert (meist inkrementiert bzw. dekrementiert), bis eine Abbruchbedingung erfüllt ist. Die Initialisierung, der boolesche Ausdruck und die Anweisung zur Aktualisierung der Zählvariablen müssen durch Semikolons voneinander getrennt sein.

BEISPIEL 2.82: Die Schleife gibt zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* aus

C#
```
for (int i = 1; i<=10; i++)
       listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
```

Sie können jedes der drei Elemente (Initialisierung, Abbruchbedingung, Aktualisierung) im Schleifenkopf auch weglassen, müssen sich aber dann anderweitig um Ersatz bemühen.

BEISPIEL 2.83: Eine äquivalente Version des Vorgängerbeispiels

C#
```
int i = 1;                              // Ersatz für Initialisierung der Zählvariablen
for (; i<=10; )
{
    listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
    i++;                                // Ersatz für Aktualisierung der Zählvariablen
}
```

while-Schleife

Bei diesem Schleifentyp steht die Organisation einer Zählervariablen nicht im Mittelpunkt, wodurch eine etwas flexiblere Programmierung möglich wird. In Abhängigkeit davon, ob die Abbruchbedingung am Schleifenanfang oder an deren Ende kontrolliert wird, spricht man auch von *kopfgesteuerten* bzw. *fußgesteuerten* Schleifen. Die *while*-Schleife ist – ebenso wie die *for*-Schleife – kopfgesteuert.

BEISPIEL 2.84: Ein völlig identisches Resultat wie obige for-Schleifen

C#
```
int i = 1;
while (i <= 10)
{
   listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
   i++;
}
```

do-Schleife

Dieser dritten Schleifentyp ähnelt der *while*-Schleife, allerdings wird die Abbruchbedingung erst am Ende getestet.

BEISPIEL 2.85: Eine weitere äquivalente Version der Vorgängerbeispiele

C#
```
int i = 1;
do
{
  listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
  i ++;
}
while (i <= 10);
```

Das komplette Programm finden Sie im Praxisbeispiel

► 2.8.3 Schleifenanweisungen verstehen

Wozu braucht man die *do*-Schleife überhaupt, wenn man doch mit der *while*-Schleife zum gleichen Ergebnis kommt?

Die Antwort: Manchmal brauchen Sie vielleicht eine Schleife, bei der die Anweisungen im Schleifenkörper mindestens einmal ausgeführt werden. In diesem Fall dürfte der Unterschied deutlich sein.

BEISPIEL 2.86: Ersetzen Sie die erste Anweisung in beiden Vorgängerbeispielen durch

C#
```
int i = 11;
```

Während die *do*-Schleife eine einzige Zeile ausgibt, bleibt die *ListBox* bei der *while*-Schleife leer.

Vorzeitiges Verlassen einer Schleife

Genauso wie Sie mit *break* eine Verzweigung verlassen können, ist damit auch ein vorzeitiger Abbruch einer Schleifenanweisung möglich.

BEISPIEL 2.87: Vorzeitiges Verlassen einer Schleife

C#

Auch dieser Code gibt zehnmal untereinander einen Text in einer *ListBox* aus. Statt aber die Abbruchbedingung im Schleifenkopf festzulegen, wird sie in den Schleifenkörper verlagert.

```
for (int i = 1; ; i++)
{
   listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
   if (i == 10) break;            // Abbruchbedingung testen
}
```

Im Gegensatz zu *break* bewirkt *continue*, dass die Abbruchbedingung sofort (also nicht erst beim nächsten Schleifeneintritt) ausgewertet wird.

HINWEIS: Da die *continue*-Anweisung zu schwer auffindbaren Programmierfehlern führen kann, sollten Sie sie nur in Ausnahmefällen verwenden.

2.6 Benutzerdefinierte Datentypen

Sie sind als Programmierer nicht nur auf die einfachen Datentypen *int, float, ...* angewiesen, sondern können auch eigene Datentypen kreieren. Interessant sind vor allem

- Enumerationen und
- strukturierte Datentypen.

Beide bauen auf den einfachen Datentypen auf und sind – ebenfalls wie diese – so genannte *Wertetypen*, die im Stack des Computers gespeichert werden.

2.6.1 Enumerationen

Mit dem *enum*-Schlüsselwort erstellen Sie einen Enumtyp, dessen mögliche Werte auf eine bestimmte Menge symbolischer Namen beschränkt ist. Häufig werden Sammlungen von miteinander verwandten Konstanten in Enumerationen zusammengefasst.

Deklaration

Der *enum*-Datentyp wird ähnlich wie ein strukturierter Datentyp deklariert.

SYNTAX:
```
enum Bezeichner : Datentyp
{
  Feld1 = Wert1;
  Feld2 = Wert2;
  ...
}
```

Alle in der Enumeration enthaltenen Konstanten müssen vom gleichen Datentyp sein, zulässig sind nur die acht Integer-Typen *byte*, *sbyte, short*, *ushort, int, uint, long* und *ulong*. Falls die Angabe des Datentyps weggelassen wird, handelt es sich automatisch um *int*-Konstanten.

BEISPIEL 2.88: Eine Enumeration für die Monate eines Jahres

C#
```
enum Monate : byte
{Januar = 1, Februar, März, April, Mai, Juni,
 Juli, August, September, Oktober, November, Dezember}
```

Jedem Element der Enumeration ist ein bestimmter Zahlenwert zugeordnet, der standardmäßig mit 0 beginnt. Falls Sie das ändern wollen, so müssen Sie den Wert explizit zuordnen. Wenn Sie das nur für den ersten Wert tun, so erhalten die Nachfolger vom Compiler automatisch den nächsthöheren Wert (siehe obiges Beispiel).

Zugriff

Wenn ein Enumtyp deklariert ist, können Sie ihn wie jeden anderen Datentyp verwenden. Das heißt, Sie deklarieren damit auf gewohnte Weise Variablen oder Konstanten, denen Sie dann Werte zuweisen.

BEISPIEL 2.89: Deklarieren der Konstanten *ersterMonat* vom Enumtyp *Monate*

```
C# const Monate ersterMonat = Monate.Januar;
```

BEISPIEL 2.90: Deklarieren von zwei Variablen des Enumtyps *Monate*

```
C# Monate monat1, monat2;
   monat1 = Monate.Januar;
   monat2 = monat1 + 1;
   MessageBox.Show(monat2.ToString());    // zeigt Februar
```

Um nicht den Namen, sondern die Integer-Zahl des Elements zu ermitteln, müssen Sie eine Typkonvertierung vornehmen.

BEISPIEL 2.91: Der folgende Code wird an das Vorgängerbeispiel angehängt

```
C# byte m = (byte) monat2;
   MessageBox.Show(m.ToString());        // zeigt 2
```

2.6.2 Strukturen

Mit dem *struct*-Schlüsselwort definieren Sie komplexe Datentypen, die ein oder auch mehrere Element(e) enthalten dürfen. In diesem Abschnitt wollen wir aber Strukturen nicht tiefgründiger behandeln, sondern nur einen ersten Einblick gewähren.

Deklaration

Vom Prinzip her entspricht die Definition einer *Struktur* der einer *Klasse*, die allerdings zu den so genannten *Verweis*- oder *Referenztypen* gehört (siehe Kapitel 3).

Eine einfache Struktur hat folgenden Aufbau:

```
SYNTAX: struct Bezeichner
        {
           Modifizierer Datentyp feld1;
           Modifizierer Datentyp feld2;
           ...
        }
```

Beachten Sie:

- Die *struct*-Anweisung ist innerhalb von Methoden unzulässig und nur auf globaler Ebene anwendbar.
- Damit die Elemente einer Struktur von außerhalb sichtbar sind, muss der *public*-Modifizierer vorangestellt werden[1].

[1] Diese Darstellung ist etwas vereinfacht, die ausgereifte Programmierung verlangt auch hier, genauso wie bei Klassen, das Prinzip der Kapselung.

BEISPIEL 2.92: Eine Struktur

C# Um in einer Variablen zur Personenbeschreibung neben dem Namen auch noch das Alter zu erfassen, definieren Sie einen neuen Datentyp:

```
struct Person
{
  public string vorName, nachName;
  public byte alter;
}
```

HINWEIS: Denken Sie immer daran, dass allein mit der Definition eines Datentyps noch keine Variable dieses Typs existiert! Diese muss – wie jede andere Variable auch – erst noch deklariert werden.

BEISPIEL 2.93: Sie erzeugen zwei Variablen des oben definierten Datentyps *Person*

C#
```
Person person1, person2;
```

Datenzugriff

Um auf den Wert einer Strukturvariablen zuzugreifen, müssen Name und Element durch einen Punkt (so genannter *Qualifizierer*) voneinander getrennt sein[1].

BEISPIEL 2.94: Fortsetzung des Vorgängerbeispiels

C#
```
int a; string name;
person1.vorName = "Max";        // Initialisieren  (Schreibzugriff)
person1.nachName = "Müller";    // dto.
person1.alter = 50;             // dto.
a = person1.alter;               // Lesezugriff
name = person1.vorName + "  " + person1.nachName;          // "Max Müller"
```

HINWEIS: Beim Lesezugriff auf nicht initialisierte Felder erhalten Sie einen Compilerfehler.

BEISPIEL 2.95: Das Feld *alter* von *person2* wurde nicht initialisiert (siehe Bemerkungen)

C#
```
a = person2.alter;    // Fehler!
```

Man kann natürlich nicht nur, wie eben beschrieben, auf die einzelnen Felder einer Strukturvariablen, sondern auch auf die Variable insgesamt zugreifen.

BEISPIEL 2.96: Die Variable *person1* wird "geklont"

C#
```
person2 = person1;
```

[1] Dies entspricht der generellen Schreibweise beim Zugriff auf Objekteigenschaften.

Struktur versus Klasse

Wie bereits bekannt, haben Strukturen viele Ähnlichkeiten mit Klassen. Genauso wie diese können sie beispielsweise über einen oder mehrere (überladene) Konstruktoren verfügen, mit denen die Felder initialisiert werden können. Aber es gibt da mehrere wesentliche Unterschiede, unter anderem können Sie für eine Struktur selbst keinen Standard-Konstruktor (das ist einer ohne Parameter, d.h. mit leeren Klammern) erstellen, denn dies wird immer vom Compiler erledigt, der die Felder dann (je nach Datentyp) mit den Werten *0*, *null* oder *false* initialisiert.

BEISPIEL 2.97: Die Strukturvariable *person1* wird mit ihrem Standard-Konstruktor erzeugt und initialisiert

C#
```
Person person1 = new Person();
int a = person1.alter;                // a erhält automatisch den Wert 0
```

Und noch ein wesentlicher Unterschied zu Klassen und anderen Verweistypen:

HINWEIS: Bei Strukturvariablen spielt sich nach wie vor alles auf dem Stack ab (auch das Instanziieren mit *new*), sie sind Wertetypen und für diese ist der Heap tabu!

2.7 Nutzerdefinierte Methoden

Methoden kapseln wiederverwendbaren Programmcode und erleichtern somit nicht nur die Arbeit des Programmierers, sondern tragen auch im erheblichen Maß zur Übersichtlichkeit des Programmcodes bei.

Jede Methode hat einen Namen und einen Körper. Letzterer enthält die beim Methodenaufruf auszuführenden Anweisungen.

Grundsätzlich unterscheiden wir zwischen

- Methoden **mit** Rückgabewert
- Methoden **ohne** Rückgabewert

Methoden **mit** Rückgabewert sind Ihnen sicherlich aus anderen Programmiersprachen[1] als *Funktionen* ein Begriff, während eine Methode **ohne** Rückgabewert als *Prozedur, Sub* oder einfach nur als *Unterprogramm* bezeichnet wird.

In diesem Abschnitt wollen wir Methoden noch nicht als öffentliche (*public)* Mitglieder einer Klasse verstehen, wie sie in der OOP das Verhalten von Objekten definieren, denn dazu kommen wir erst im Kapitel 3. Vorerst behandeln wir sie lediglich als private Klassenmitglieder.

HINWEIS: Für das Verständnis der Parameterübergabe an Methoden ist es wichtig, dass Sie den Unterschied zwischen Werte- und Referenztypen kennen (siehe 2.2.2).

[1] z B. Visual Basic oder Delphi

2.7.1 Methoden mit Rückgabewert

Deklaration

Die Deklaration ist ähnlich wie bei einer normalen Variablen, nur dass an den Namen der Methode in Klammern die Parameterliste angefügt wird (kann auch leer sein).

SYNTAX:
```
Datentyp Name (Parameterliste)
{
   // Code definieren
   // .....
   // ...
   return (Rückgabewert)
}
```

Die *Parameterliste* ist eine durch Kommata separierte Liste mit der Aufzählung der einzelnen Parameter:

SYNTAX: `([ref|out] Datentyp Parameter, [ref|out] Datentyp Parameter, ...)`

HINWEIS: Zur Bedeutung der optionalen Übergabe mit *ref* bzw. *out* siehe die Abschnitte 2.7.3 und 2.7.4.

Der Methodenkörper ist durch eine öffnende und eine schließende geschweifte Klammer eingegrenzt.

Als Rückgabewert kommt natürlich nur der im Methodenkopf deklarierte Datentyp in Frage. Den Rückgabewert, der auch ein Ausdruck sein kann, schreiben Sie in Klammern hinter die *return*-Anweisung.

BEISPIEL 2.98: Funktion zur Berechnung des Kugelgewichts.

C#

Übergabeparameter sind der Radius *ra* und das spezifische Gewicht *sg*.

```
private double Kugel(double ra, double sg)
{
   double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
   return(sg * vol);
}
```

HINWEIS: Schreiben Sie die *return*-Anweisung immer an das Ende des Methodenkörpers, denn alle dahinter stehenden Anweisungen werden nicht mehr ausgeführt.

Aufruf

Beim Aufruf einer Methode muss die *Signatur* der Parameterliste sorgfältig beachtet werden, d.h., die Reihenfolge der Argumente und ihr Datentyp müssen zur deklarierten Parameterliste passen.

Im aufrufenden Code muss der Rückgabewert einer Variablen zugewiesen werden, die den gleichen Datentyp wie die Methode hat.

BEISPIEL 2.99: Obige Funktion wird mit einer eisernen Kugel von 20 cm Durchmesser getestet.

C#
```
double r = 10, s = 7.87;           // Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
double gew = Kugel(r, s);          // Aufruf
MessageBox.Show(gew.ToString("#,#0.000 Gramm"));   // Gewicht der Kugel ist 32.965,779 Gramm
```

HINWEIS: Bei **jedem** Methodenaufruf müssen Sie die runden Klammern **()** angeben, auch wenn die Methode keine Parameter besitzt!

2.7.2 Methoden ohne Rückgabewert

Deklaration

Soll die Methode keinen Wert zurückgeben, so setzen Sie anstelle des Datentyps das *void*-Schlüsselwort ein.

SYNTAX:
```
void Name (Parameterliste)
{
   // Code definieren
   // .....
}
```

Eine *void*-Methode arbeitet oft mit *Feldern* zusammen. Unter Feldern verstehen wir in C# globale Variablen, die auf Klassenebene deklariert wurden (im Unterschied zu den lokalen Variablen im Methodenkörper).

BEISPIEL 2.100: Anwenden einer *void*-Methode zum Berechnen des Kugelgewichts.

C#

Deklarationen auf Klassenebene:

```
private double gew;      // dieser globalen Variablen wird das Ergebnis zugewiesen

private void Kugel(double ra, double sg)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  gew = sg * vol;        // Feld wird mit dem Ergebnis gefüttert
}
```

Aufruf

Im Unterschied zum Aufruf einer Methode mit Rückgabewert handelt es sich diesmal um keine direkte Zuweisung des Rückgabewerts, da dieser der globalen Variablen *gew* entnommen wird.

BEISPIEL 2.101: Unsere Methode wird nun ebenfalls mit der legendären Eisenkugel getestet.

C#

```
double r = 10, sg = 7.87;          // Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
Kugel(r, sg);                      // Aufruf
MessageBox.Show(gew.ToString("#,#0.000 Gramm"));   //  zeigt "32.965,779 Gramm"
```

HINWEIS: Obwohl Sie in einer *void*-Methode die *return*-Anweisung nicht brauchen, können Sie diese trotzdem zum vorzeitigen Verlassen der Methode verwenden. Dazu schreiben Sie einfach *return* gefolgt von einem Semikolon.

2.7.3 Parameterübergabe mit ref

Bei jedem Methodenaufruf werden – je nach Länge der Parameterliste – kein, ein oder auch mehrere Argument(e) an die Parameter der Methode übergeben. Handelt es sich hierbei um einen Wertetyp, so wird der entsprechende Parameter automatisch zu einer Kopie des Arguments.

Wenn Sie aber bei der Methodendeklaration dem Parameter das Schlüsselwort *ref* voranstellen, so wird nicht eine Kopie, sondern der Zeiger auf die Speicherplatzadresse des Arguments übermittelt. Im Körper der aufgerufenen Methode wird es dadurch möglich, am Wert des Arguments "herumzudoktern", denn sämtliche hier vorgenommenen Änderungen am Wert des Parameters wirken sich auf das Argument aus. Auf diese Weise können Methoden ihre Ergebnisse ohne Verwendung globaler Variablen direkt an die aufrufenden Argumente zurückliefern.

HINWEIS: Wenn Sie ein Argument an einen *ref*-Parameter übergeben, müssen Sie beim Aufruf ebenfalls das *ref*-Schlüsselwort voranstellen.

BEISPIEL 2.102: Parameterübergabe mit *ref*

C#

Bei dieser Methode zur Kugelgewichtsbestimmung wurde die Übergabeart des zweiten Parameters in *ref* geändert. Dadurch wird es möglich, über diesen Parameter auch das Berechnungsergebnis zu übertragen.

```
private void Kugel(double ra, ref double g)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  g = g * vol;        // g zeigt auf das Ergebnis
}
```

Den Parameter *g* haben wir hier bewusst nicht mit *sg* (für spezifisches Gewicht) bezeichnet, weil er nach dem Aufruf der *Kugel*-Methode auf das Gewicht verweist, also dann z.B. *gew* heißen müsste.

Test

Testen der Methode:

```
double r = 10, g = 7.87;
Kugel(r, ref g);
MessageBox.Show(g.ToString("#,#0.000 Gramm"));     // zeigt "32.965,779 Gramm"
```

Noch einmal – aber bitte langsam

Der *ref*-Parameter hat nur für Wertetypen (dazu gehören z.B. die einfachen Datentypen wie *int*, *double*) eine relevante Bedeutung. Mit der standardmäßigen Parameterübergabe (also ohne *ref*) wird der Wert des übergebenen Parameters in eine vom Compiler erzeugte lokale Variable des Methodenkörpers kopiert. Die aufrufende Variable (das Argument) und die lokale Variable im Inneren der Methode (der Parameter) haben danach nichts mehr miteinander zu tun, eine Veränderung des Wertes der lokalen Variablen bleibt ohne Rückwirkung auf das Argument, sodass ein "Zurückgeben" von Ergebnissen – wie im obigen Beispiel gezeigt – nicht möglich ist.

Wenn Sie das Argument (einen Wertetyp) hingegen mit *ref* übergeben, so verfügt die aufgerufene Methode nur scheinbar über eine eigene lokale Variable. In Wirklichkeit wird mit dem ursprünglichen Argument weitergearbeitet, eine "Entkopplung" hat also nicht stattgefunden.

HINWEIS: Im Interesse der Fehlersicherheit Ihrer Programme sollten Sie *ref*-Parameter möglichst sparsam einsetzen und ansonsten lieber mit Kopien arbeiten. *ref* sollten Sie nur dann verwenden, wenn Sie tatsächlich Werte von Wertetypen "nach außen hin" verändern wollen.

2.7.4 Parameterübergabe mit out

Das Schlüsselwort *out* hat eine sehr ähnliche Bedeutung wie *ref*. Also wird auch hier der Wert des Arguments nicht in den Parameter kopiert, sondern der Parameter wird lediglich zu einem Alias des Arguments.

Einziger Unterschied zwischen *ref* und *out* ist, dass eine Methode einem *ref*-Parameter einen Wert zuweisen **kann**, einem *out*-Parameter hingegen einen Wert zuweisen **muss**.

HINWEIS: Ein Argument, das an einen *ref*-Parameter übergeben wird, muss zuvor initialisiert werden. Dies ist beim *out*-Parameter nicht erforderlich.

BEISPIEL 2.103: Parameterübergabe mit *out*

C#

Dieses Beispiel würde mit *ref* nicht funktionieren, da der beim Aufruf übergebene Parameter *g* nicht initialisiert ist.

```
void Kugel(double ra, double sg, out double gew)
{
  double vol = 4 / 3F * Math.PI * Math.Pow(ra, 3);
  gew = sg * vol;        // gew zeigt auf das Ergebnis
}
```

Test

Der Aufruf:

```
double r = 10, s = 7.87, g;        // g ist nicht initialisiert!
Kugel(r, s, out g);
MessageBox.Show(g.ToString("#,#0.000 Gramm"));    // zeigt "32.965,779 Gramm"
```

2.7.5 Methodenüberladung

Wenn mehrere gleichnamige Methoden im gleichen Gültigkeitsbereich ohne Namenskonflikt friedlich nebeneinander existieren, so spricht man von *überladenen* Methoden. Die Unterscheidung wird vom Compiler anhand der Signatur getroffen. Unter Signatur verstehen wir bekanntlich die Reihenfolge der übergebenen Parameter und deren Datentyp.

HINWEIS: Überladene Methoden müssen immer eine unterschiedliche Signatur haben!

BEISPIEL 2.104: Methodenüberladung

C#

Die *Kugel*-Funktion steht in zwei überladenen Versionen zur Verfügung. Die erste Version verlangt als Parameter den Radius und das spezifische Gewicht.

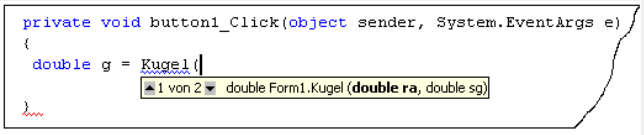
```
double Kugel(double ra, double sg)
{
    double vol = 4 / 3.0 * Math.PI * Math.Pow(ra, 3);
    return(sg * vol);
}
```

Eine zweite Version soll nur das Volumen der Kugel berechnen.

```
double Kugel(double ra)
{
   double vol = 4 / 3.0 * Math.PI * Math.Pow(ra, 3);
   return(vol);
}
```

Test

Wenn Sie jetzt die Methode verwenden wollen, werden Ihnen in der Visual Studio-Entwicklungsumgebung automatisch beide überladenen Versionen angeboten:



Der Aufruf ist entweder so

```
double g = Kugel(10, 7.87);          // liefert  32.965,779
```

oder so möglich:

```
double v = Kugel(10);               // liefert  4.188,790
```

Bemerkungen

- Unterscheiden sich die Deklarationen zweier Methoden nur in der Verwendung von *ref* bzw. *out*, erfolgt ebenfalls eine Überladung.
- Das Überladen des Rückgabewerts einer Methode ist nicht erlaubt.

- Wie Sie bestimmt schon bei der Arbeit im Codefenster von Visual Studio bemerkt haben, werden für die meisten Methoden mehrere Überladungen angeboten. Ein typisches Beispiel hierfür ist die *Show*-Methode der *MessageBox* mit insgesamt 12 (!) Überladungen.

2.7.6 Optionale Parameter

Ein optionaler Parameter gestattet es Ihnen, einem Parameter einen Standardwert zuzuweisen. Beim Aufruf der Methode bleibt es Ihnen überlassen, für diesen Parameter ein Argument zu übergeben oder nicht.

Das folgende Beispiel zeigt, wie Sie in früheren C#-Versionen optionale Parameter mittels Methodenüberladungen simulieren konnten.

BEISPIEL 2.105: Zwei Überladungen einer Methode

C#
```
public static string Hallo()              // erste Überladung
{
    return "Hallo Du!";
}

public static string Hallo(string name)    // zweite Überladung
{
    return "Hallo " + name + "!";
}
```

Bei Verwendung dieses Codes erscheint der *name*-Parameter wie ein optionaler Parameter, denn die folgenden beiden Aufrufe funktionieren:

```
string grüssDich = Hallo();
string grüssFernando = Hallo("Fernando");
```

Wenn Sie Methoden mit mehr Parametern hätten, müssten Sie noch mehr Überladungen bereitstellen. Vielen Entwicklern gefällt dies nicht, da dadurch der Code aufquillt wie Hefeteig.

Das folgende Beispiel zeigt einen optionalen Parameter, welcher die obigen zwei Überladungen ersetzt:

BEISPIEL 2.106: Methode mit einem optionalen Parameter

C#
```
public static string Hallo(string name = "Du")
{
    return "Hallo " + name + "!";
}
```

Wie zu sehen ist, erfordert die Aufrufsyntax für einen optionalen Parameter lediglich das Zuweisen eines Standardwerts (*default value*). Falls beim Aufruf kein Wert übergeben wird, wird der Standardwert genommen.

Auch das folgende Beispiel aus der Office-Automation zeigt, wie man mittels optionaler Parameter den Code vereinfachen kann.

BEISPIEL 2.107: Die *SaveAs*-Methode für ein Excel-Worksheet in klassischer Schreibweise:

C#
```
wkBook.SaveAs("myfile.xls", Missing.Value, Missing.Value,
              Missing.Value, Missing.Value, Missing.Value,
              Excel.XlSaveAsAccessMode.xlShared,
              Missing.Value, Missing.Value, Missing.Value, Missing.Value, Missing.Value);
```

Nur zwei Argumente werden im obigen Beispiel bereitgestellt, die übrigen sind *Missing.Value*.

Jetzt die vereinfachte Version mit optionalen Parametern:

```
wkBook.SaveAs("myfile.xls", Excel.XlSaveAsAccessMode.xlShared);
```

Wenn Sie bislang für Methoden wie *SaveAs* immer kostbare Zeit verschwenden mussten, um die exakte Position eines Parameters herauszufinden, werden Sie die Kürze obigen Codes zu schätzen wissen[1].

2.7.7 Benannte Parameter

Eng verwandt mit den optionalen Parametern sind die benannten Parameter. Einer ihrer Hauptvorzüge ist die Auflösung von Mehrdeutigkeiten. Benannte Parameter erkennt man nicht anhand der Methodendeklaration, sondern lediglich an der Syntax des Methodenaufrufs.

HINWEIS: Benannte Parameter tragen beim Aufruf denselben Namen wie die entsprechenden Parameter in der Methodendeklaration, wobei ein Doppelpunkt angehängt wird.

BEISPIEL 2.108: Eine Methode mit mit drei Parametern

C#
```
public static void addBuch(string titel, string autor = "", DateTime? termin = null)
{
    // noch nicht implementiert
}
```

Der Parameter *titel* ist nicht optional, *autor* hingegen ist optional, beide sind vom Datentyp *string*. Der letzte Parameter zeigt, dass man als Standardwert einem Parameter auch *null* zuweisen kann.

Der Aufruf der Methode mit benannten Parametern:

```
addBuch(autor: "Doberenz und Gewinnus",  termin: DateTime.Now, titel: "Hallo Fernando!");
```

Bewusst haben wir hier die Reihenfolge der übergebenen Parameter verändert, da diese bei benannten Parametern keine Rolle spielt.

Benannte Parameter sind besonders dann nützlich, wenn Sie mehrere optionale Parameter gleichen zu übergebenden Typs haben.

[1] Im Fall des zweiten Arguments käme auch ein benannter Parameter (siehe folgender Abschnitt) infrage. Da aber die Parameterliste nur ein Argument enthält, ist eine Benennung nicht erforderlich.

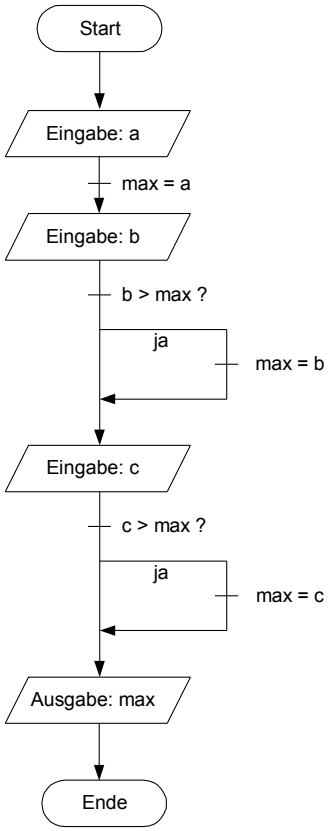
2.8 Praxisbeispiele

2.8.1 Vom PAP zur Konsolenanwendung

Dieses ausgesprochene Einsteiger-Beispiel erläutert die Umsetzung eines Programmablaufplans (PAP) in eine klassische Konsolenanwendung. Es sind nacheinander drei positive ganze Zahlen einzugeben. Das Programm soll die größte der drei Zahlen ermitteln und das Ergebnis anzeigen!

Programmablaufplan

Der nachfolgend abgebildete PAP zeigt die Berechnungsvorschrift (Algorithmus). Sie erkennen hier die typische EVA-Grundstruktur, bei der die Anweisungen in der Reihenfolge **E**ingabe, **V**erarbeitung, **A**usgabe ausgeführt werden.



Programmierung

Beim Schreiben der Befehle (z.B. mit dem Editor aus dem Windows-Zubehör) gibt obiger PAP eine nützliche Orientierung:

```
using System;
class Maximum3
{
   static void Main()
   {
      Console.WriteLine("Maximum von drei Zahlen");          // Überschrift
      Console.WriteLine();                                   // Leerzeile
      int a, b, c, max;                                      //  Variablendeklaration

      Console.WriteLine("Geben Sie die erste Zahl ein!");
      a = Convert.ToInt32(Console.ReadLine());               // Eingabe a
      max = a;                                               // Initialisieren von max
      Console.WriteLine("Geben Sie die zweite Zahl ein!");

      b = Convert.ToInt32(Console.ReadLine());               // Eingabe von b
      if (b > max) max = b;                                  // Bedingung
      Console.WriteLine("Geben Sie die dritte Zahl ein!");

      c = Convert.ToInt32(Console.ReadLine());               // Eingabe c
      if (c > max) max = c;                                  // Bedingung
      Console.WriteLine("Das Maximum ist " + max.ToString());      // Ergebnisausgabe
      Console.ReadLine();                        // Programm wartet auf <Enter>, um zu beenden
   }
 }
```

Speichern Sie die Textdatei als *Maximum3.cs* ab.

Kompilieren

Die Vorgehensweise entspricht exakt dem Abschnitt 1.2 im Einführungskapitel, Sie müssen also zunächst diverse Vorbereitungen treffen (Umgebungsvariable für den C#-Compiler hinzufügen, Datei *cmd.exe* in das Anwendungsverzeichnis kopieren), um bequem kompilieren zu können.

Geben Sie dann den folgenden Text an der Kommandozeile ein:

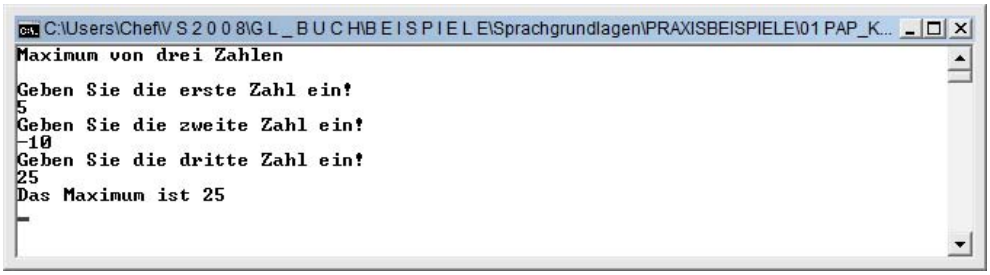
```
csc /t:exe Maximum3.cs
```

Haben Sie beim Eintippen des Quellcodes keine Fehler gemacht, so dürfte das Kompilieren anstandslos verlaufen.

Im Projektverzeichnis finden Sie nun die Datei *Maximum3.exe* vor.

Test

Starten Sie *Maximum3.exe* durch Doppelklick!



HINWEIS: Durch Drücken der *Enter*-Taste beenden Sie die Anwendung.

2.8.2 Ein Konsolen- in ein Windows-Programm verwandeln

Eine Windows-Anwendung ist natürlich wesentlich attraktiver als eine Konsolen-Applikation und schließlich wollen Sie ja zukünftig mit dem Komfort von Visual Studio statt mit einem simplen Texteditor arbeiten!

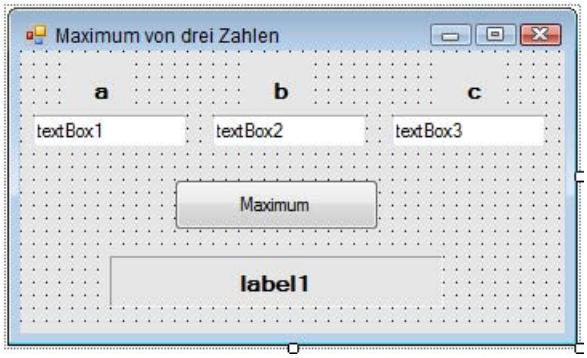
Ziel dieses Beispiels soll es sein, das im Vorgängerbeispiel erstellte Konsolen-Programm in eine "richtige" Windows Forms-Applikation zu verwandeln.

Oberfläche

Starten Sie Visual Studio und öffnen Sie ein neues Projekt (Projekttyp: "Visual C#", Vorlage: "Windows Forms-Anwendung"). Geben Sie als Namen z.B. "Maximum3" ein.

Mit *F4* holen Sie das Eigenschaftenfenster in den Vordergrund und stellen damit die *Text*-Eigenschaft (das ist die Beschriftung der Titelleiste) des Startformulars *Form1* neu ein: "Maximum von drei Zahlen".

Vom Werkzeugkasten (*Strg+Alt+X*) ziehen Sie die Steuerelemente (3 mal *TextBox*, 1 mal *Button*, 4 mal *Label*) gemäß folgender Abbildung auf *Form1* und stellen auch hier bestimmte *Text*-Eigenschaften neu ein:



Quelltext

Durch einen Doppelklick auf *button1* wird automatisch das Codefenster der (partiellen) Klasse *Form1* mit dem bereits vorbereiteten Rahmencode des *Click*-Eventhandlers geöffnet. In diesem Zusammenhang ein für den Einsteiger wichtiger Hinweis, der auch für die Zukunft gilt:

HINWEIS: Sie sollten den Rahmencode der Eventhandler nur in Ausnahmefällen selbst eintippen. Lassen Sie sich stattdessen den Rahmencode durch die Visual Studio-Entwicklungsumgebung generieren (falls es sich um das Standardereignis handelt durch Doppelklick auf das Steuerelement, ansonsten über die "Ereignisse"-Seite des Eigenschaftenfensters)!

```
using System;
using System.Windows.Forms;
...

namespace Maximum3
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            // Hier müssen Ihre C#-Anweisungen eingefügt werden!
        }
    }
}
```

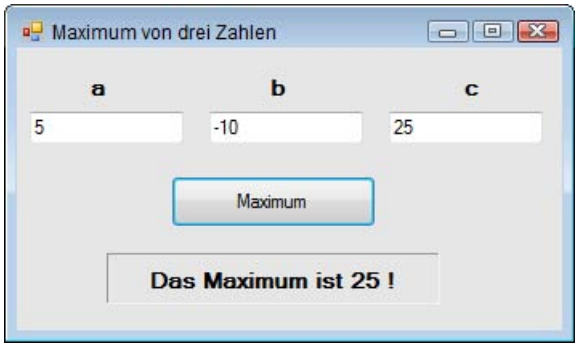
Füllen Sie den zunächst leeren Körper des Eventhandlers mit den erforderlichen Anweisungen aus, sodass der komplette Eventhandler schließlich folgendermaßen aussieht:

```
private void button1_Click(object sender, EventArgs e)
{
    int a = Convert.ToInt32(textBox1.Text);     // Eingabe a
    int max = a;                                // Initialisieren von max
    int b = Convert.ToInt32(textBox2.Text);     // Eingabe b
    if (b > max) max = b;                       // Bedingung
    int c = Convert.ToInt32(textBox3.Text);     // Eingabe c
    if (c > max) max = c;                       // Bedingung
    label1.Text = "Das Maximum ist " +  max.ToString() + " !";      // Ergebnisausgabe
}
```

HINWEIS: Beim Vergleich mit der Konsolenanwendung erkennen Sie, dass Ein- und Ausgabe deutlich einfacher geworden sind!

Test

Nachdem Sie das Projekt abgespeichert haben, kompilieren und starten Sie das Programm mit der *F5*-Taste (oder durch Klick auf die entsprechende kleine Schaltfläche mit dem grünen Dreieck in der Symbolleiste von Visual Studio):



Bemerkungen

Neben dem attraktiveren Outfit einer Windows-Anwendung schlagen auch noch weitere Vorteile gegenüber der tristen Konsolenanwendung deutlich zu Buche:

- So ist z.B. die Reihenfolge der Zahleneingaben ohne Bedeutung und
- Sie können bequem mittels *Tab*-Taste zwischen den Steuerelementen wechseln.

2.8.3 Schleifenanweisungen verstehen

C# offeriert Ihnen ein reichhaltiges Angebot an Schleifenanweisungen. Da der Umgang mit ihnen zum Einmaleins des Programmierens gehört, demonstriert Ihnen das vorliegende Beispiel die prinzipielle Anwendung der wichtigsten Schleifentypen (außer *foreach*-Schleife).

Ziel soll es sein, zehnmal untereinander einen Text in einer *ListBox* auszugeben, wobei fünf verschiedene Schleifenkonstruktionen gegenübergestellt werden.

Oberfläche

Alles, was Sie zum Testen brauchen, ist das mit einer *ListBox* und einigen *Button*s bestückte Startformular *Form1* (siehe Laufzeitabbildung am Schluss).

Quellcode

```
public partial class Form1 : Form
{   ...
```

Wir beginnen mit der altbekannten *for*-Schleife:

```
    private void button1_Click(object sender, EventArgs e)
    {
```

```
        for (int i = 1; i<=10; i++)
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
    }
```

Eine der möglichen Modifikationen, wo Sie sich selbst um die Verwaltung der Zählvariablen kümmern müssen:

```
    private void button2_Click(object sender, EventArgs e)
    {
        int i = 1;                    // Ersatz für Initialisierung der Zählvariablen
        for (; i<=10; )
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i++;                        // Ersatz für Aktualisierung der Zählvariablen
        }
    }
```

Die *while*-Schleife ist kopfgesteuert:

```
    private void button3_Click(object sender, EventArgs e)
    {
        int i = 1;
        while (i <= 10)
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i++;
        }
    }
```

Fußgesteuert hingegen gibt sich die *do-while*-Schleife:

```
    private void button4_Click(object sender, EventArgs e)
    {
        int i = 1;
        do
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            i ++;
        }
        while (i <= 10);
    }
```

Das vorzeitige Verlassen der Schleife mittels *break*:

```
    private void button5_Click(object sender, EventArgs e)
    {
        for (int i = 1; ; i++)
        {
            listBox1.Items.Add(i.ToString() + " Viele Wege führen nach Rom!");
            if (i == 10) break;                                  // Abbruchbedingung
        }
    }
```

Eher nebensächlich ist das Löschen der *ListBox*:

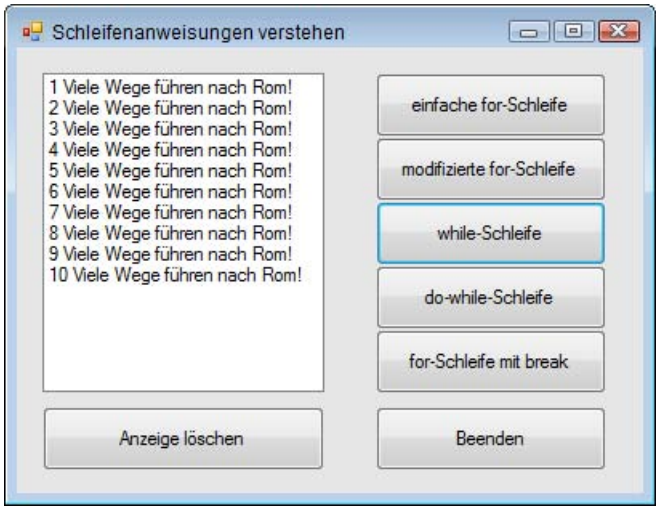
```
    private void button6_Click(object sender, EventArgs e)
    {
        listBox1.Items.Clear();
    }
```

Das Beenden des Programms:

```
    private void button7_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Test

Alle fünf Schleifenvarianten produzieren ein absolut identisches Ergebnis:



2.8.4 Benutzerdefinierte Methoden überladen

In C# spricht man nicht mehr im klassischen Sinn von Funktionen und Prozeduren, sondern nur noch von Methoden mit Rückgabewert und von Methoden ohne Rückgabewert (*void*). Beschränkt man sich auf private Methoden (Aufruf innerhalb einer Klasse), so spielen diese quasi die Rolle von "Unterprogrammen", sind also strenggenommen keine Methoden mehr im Sinne der OOP. Trotzdem können auch solche Methoden überladen werden, d.h., in mehreren gleichnamigen Versionen nebeneinander existieren, die sich nur durch ihre Signatur unterscheiden.

Das vorliegende, durchaus auch praxistaugliche, Programm demonstriert drei verschiedene Überladungen der bekannten Formel zur Berechnung des Gewichts einer Kugel:

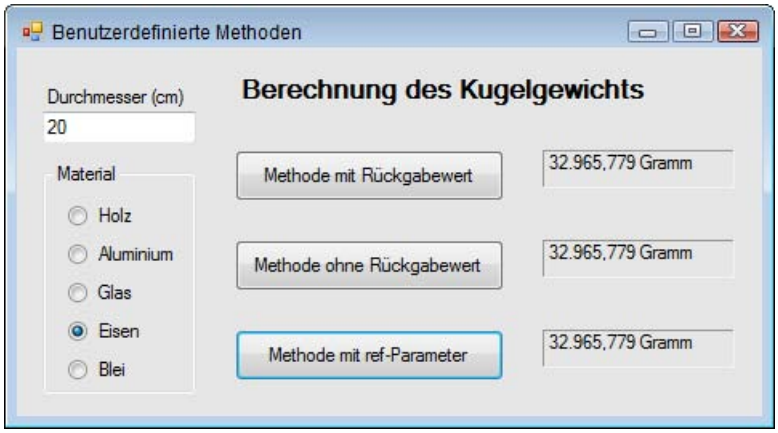
$$G = \frac{4}{3} * \pi * r^3 * \gamma$$

G = Gewicht (Gramm)
r = Radius (Zentimeter)
π = Pi = 3,14159...
γ = spezifisches Gewicht (Gramm/Kubikzentimeter)

En passant wird weiteres Einsteigerwissen wie Strukturen (*struct*), *if*-Statements, *RadioButton*-Auswahl etc. vermittelt.

Oberfläche

Auf das Startformular *Form1* setzen Sie eine *GroupBox*, die fünf verschiedene *RadioButton*s enthält, mit denen ein bestimmtes Material (Holz, Aluminium, Glas, Eisen, Blei) ausgewählt wird Weiterhin werden noch eine *TextBox*, drei *Button*s und verschiedene *Label*s benötigt (siehe folgende Laufzeitabbildung).



Quellcode

```
public partial class Form1 : Form
{ ...
```

Globale Deklarationen:

```
        private struct Kugel       // kapselt Radius und spez. Gewicht einer Kugel
        {
            public double radius, sg;
        }

        private double gew;       // Rückgabewert für Variante 2

        private Kugel getKugel()  // Hilfsfunktion, liest Kugelwerte aus Eingabemaske
        {
            Kugel kug;
            kug.radius = Convert.ToDouble(textBox1.Text)/2;     // Kugelradius zuweisen
```

```
        // spezifisches Gewicht zuweisen:
        if (radioButton1.Checked) kug.sg = 1.4D;          // Holz
        else if (radioButton2.Checked) kug.sg = 2.7D;     // Alu
        else if (radioButton3.Checked) kug.sg = 3.0D;     // Glas
        else if (radioButton4.Checked) kug.sg = 7.87D;    // Eisen
        else kug.sg = 11.3D;                              // Blei
        return kug;
    }
```

Drei verschiedene Methodenüberladungen zur Berechnung des Kugelgewichts:

Variante 1:

```
    private double KugelGewicht(double ra, double sg)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(ra, 3);
        return (sg * vol);     // Rückgabe des Gewichts als Funktionswert
    }
```

Variante 2:

```
    private void KugelGewicht(Kugel kug)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(kug.radius, 3);
        gew = kug.sg * vol;    // das Gewicht wwird der globalen Variablen gew zugewiesen
    }
```

Variante 3:

```
    private void KugelGewicht(ref Kugel kug)
    {
        double vol = 4 / 3D * Math.PI * Math.Pow(kug.radius, 3);
        kug.sg = kug.sg * vol;  // der Übergabeparameter kug zeigt auf das Gewicht!
    }
```

Bei den folgenden Codierungen der Methode *Kugelgewicht* wird Ihnen die IntelliSense von Visual Studio jeweils eine Auswahl zwischen den drei Überladungen anbieten:

Test Variante 1:

```
    private void button1_Click(object sender, EventArgs e)
    {
        Kugel kug = getKugel();
        double r = kug.radius, sg = kug.sg;             // Radius und spez. Gewicht
        double gew = KugelGewicht(r, sg);               // Aufruf
        label1.Text = gew.ToString("#,#0.000 Gramm");   // lokale Variable gew
    }
```

Test Variante 2:

```
    private void button2_Click(object sender, EventArgs e)
    {
        Kugel kug = getKugel();
```

```
            KugelGewicht(kug);                              // Aufruf
            label2.Text = gew.ToString("#,#0.000 Gramm");   // globale Variable gew
        }
```

Test Variante 3:

```
        private void button3_Click(object sender, EventArgs e)
        {
            Kugel kug = getKugel();
            KugelGewicht(ref kug);                           // Aufruf (kug trägt Ergebnis)
            label3.Text = kug.sg.ToString("#,#0.000 Gramm");
        }
    }
```

Test

Geben Sie das Material und den Durchmesser der Kugel ein und lassen Sie sich das Gewicht anzeigen. Alle drei Methodenüberladungen liefern identische Ergebnisse (siehe obige Abbildung).

Kapitel 3

OOP-Konzepte

C# erlaubt es Ihnen, bereits ohne fundierte OOP-Kenntnisse objektorientiert zu programmieren! Davon haben Sie bereits vor der Lektüre dieses Kapitels, mehr oder weniger unbewusst, Gebrauch gemacht: Sie haben Ereignisbehandlungsroutinen (Eventhandler) geschrieben und den Objekten der visuellen Benutzerschnittstelle (Form, Steuerelemente) Eigenschaften zugewiesen bzw. deren Methoden aufgerufen.

Die Entwicklungsumgebung von Visual Studio erlaubt objektorientiertes Programmieren bereits mit einem Minimum an Vorkenntnissen. Das vorliegende Kapitel will etwas tiefer in die OOP-Problematik eindringen und präsentiert Ihnen neben einigen grundlegenden Ausführungen die für den Einstieg wichtigsten objektspezifischen Features von C# im Überblick.

HINWEIS: In C# haben Sie grundsätzlich die Möglichkeit, zwischen Klassen (sind Verweistypen) und Strukturen (sind Wertetypen) zu wählen. Strukturen (*struct*) wurden bereits im Sprachkapitel (Abschnitt 2.6.2) einführend behandelt, bieten allerdings noch weitaus mehr Möglichkeiten, die fast an die von Klassen heranreichen. Wir aber wollen uns im vorliegenden Kapitel ausschließlich mit Klassen und Objekten, den Grundelementen der OOP, beschäftigen.

3.1 Kleine Einführung in die OOP

In .NET ist alles ein Objekt! Viele Entwickler – insbesondere wenn sie mit "altem" Code zu kämpfen haben – tun sich immer noch ziemlich schwer mit OOP, weil ihnen die Komplexität einer vollständigen Anwendung zu hoch erscheint.

3.1.1 Historische Entwicklung

Im Unterschied zur objektorientierten ist die klassische strukturierte Programmierung ziemlich sprachunabhängig und hatte Zeit genug, um auch in den letzten Winkel der Programmierwelt vorzudringen.

Demgegenüber stand es um die Akzeptanz der objektorientierten Programmierung bis Anbruch des .NET-Zeitalters zu Beginn dieses Jahrtausends noch nicht zum Besten, das aber hat sich seitdem dramatisch geändert.

Strukturierte Programmierung

Gern bezeichnet man die strukturierte Programmierung auch als Vorläufer der objektorientierten Programmierung, obwohl dieser Vergleich hinkt. Richtig ist, dass sowohl strukturierte als auch objektorientierte Programmierung fundamentale Denkmuster[1] sind, die gleichberechtigt nebeneinander existieren.

Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der Sechzigerjahre entwickelt und lassen sich mit folgenden Stichwörtern charakterisieren: hierarchische Programmorganisation, logische Programmeinheiten, zentrale Programmsteuerung, beschränkte Datenverfügbarkeit.

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.

Gegenstand der strukturierten Programmierung ist also die bestmögliche Anordnung von Code, um dessen Transparenz, Testbarkeit und Wiederverwendbarkeit zu maximieren[2].

Dass C# eine konsequent objektorientierte Sprache ist, bedeutet noch lange nicht, dass man damit nicht auch strukturiert programmieren könnte, im Gegenteil. Im Kapitel 2, wo sich alles um die grundlegenden sprachlichen Elemente von C# dreht, haben wir uns fast ausschließlich auf dem Boden der traditionellen strukturierten Programmierung bewegt und versucht, die OOP noch weitestgehend auszuklammern.

So haben wir es größtenteils ignoriert, dass selbst die einfachen Datentypen Objekte sind, und haben z.B. statt mit Methoden mit Funktionen und Prozeduren und statt mit Klassen mit strukturierten Datentypen (*struct*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt. Statt beispielsweise globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseneigenschaften verwenden.

Um fit für die aktuellen Herausforderungen zu sein, sollten Sie deshalb – wo immer es vertretbar ist – nach objektorientierten Lösungen streben.

Objektorientiertes Programmieren

Die objektorientierte Programmierung entfaltete auf breiter Basis erst seit Ende der 80er-Jahre mit dem Beginn des Windows-Zeitalters ihre Wirkung. Sehr bekannte Vertreter objektorientierter Sprachen sind C++, Java, Smalltalk und Borland Delphi – aber auch das alte Visual Basic war bereits in vielen wesentlichen Zügen objektorientiert aufgebaut[3].

---

[1] Im Fachjargon heißt das "Paradigma".

[2] Bahnbrechendes auf diesem Gebiet leistete Prof. Niklaus Wirth mit seinen Sprachen Pascal und Modula.

[3] Mit VB.NET wurde auch Visual Basic endlich vollwertiges Mitglied der OOP-Welt.

Objektorientierte Programmierung ist ein Denkmuster, bei dem Programme als Menge von über Nachrichten kooperierenden Objekten organisiert werden und jedes Objekt Instanz einer Klasse ist.

Im Unterschied zur strukturierten Programmierung bedeutet "objektorientiert" also, dass Daten und Algorithmen nicht mehr nebeneinander existieren, sondern in Objekten zusammengefasst sind.

Während Module in der strukturierten Programmierung zwar auch Daten und Code zusammenfassen, stellen Klassen jetzt Vorlagen dar, von denen immer neue Kopien (Instanzen) angefertigt werden können. Diese Instanzen, d.h. die Objekte, kapseln den Zugriff auf die enthaltenen Daten hinter Schnittstellen (Interfaces).

Der große Vorteil der OOP ist ihre Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird vor allem dem Einsteiger, der bisher über keine bzw. wenig Programmiererfahrung verfügt, das Verständnis der OOP erleichtert.

HINWEIS: Die OOP verlangt eine Anpassung des Software-Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil des Programmierers – nicht umgekehrt!

Die OOP ist eine der wenigen Fälle, in denen der Einsteiger gegenüber dem Profi zumindest einen kleinen Vorteil besitzt: Er ist noch nicht in der Denkweise klassischer Programmiersprachen gefangen, die dazu erziehen, in Abläufen zu denken, bei denen die in der realen Welt zu beobachtenden Abläufe Schritt für Schritt in Algorithmen umgesetzt werden, etwa um betriebliche Prozesse per Programm zu automatisieren.

Die OOP entspricht hingegen der üblichen menschlichen Denkweise, indem sie z.B. reale Objekte aus der abzubildenden Umwelt identifiziert und in ihrer Art beschreibt.

3.1.2 Grundbegriffe der OOP

Bereits im Kapitel 1hatten Sie gesehen, dass Objekte durch Eigenschaften, Methoden und Ereignisse beschrieben werden. Auf diese und andere Weise überwindet das Konzept der objektorientierten Programmierung (OOP) den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung (OOP) zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

Objekt

Der Programmierer versteht unter einem *Objekt* die Zusammenfassung (Kapselung) von Daten und zugehörigen Funktionalitäten. Ein solches Softwareobjekt wird auch oft benutzt, um Dinge des täglichen Lebens für Zwecke der Datenverarbeitung abzubilden. Aber das ist nur ein Aspekt, denn Objekte sind ganz allgemein Dinge, die Sie in Ihrem Code beschreiben wollen, es sind Gruppen von Eigenschaften, Methoden und Ereignissen, die logisch zusammengehören. Als Programmierer arbeiten Sie mit einem Objekt, indem Sie dessen Eigenschaften und Methoden manipulieren und auf seine Ereignisse reagieren.

Klasse

Eine *Klasse* ist nicht mehr und nicht weniger als ein "Bauplan", auf dessen Grundlage die entsprechenden Objekte zur Programmlaufzeit erzeugt werden. Gewissermaßen als Vorlage (Prägestempel) für das Objekt legt die Klasse fest, wie das Objekt auszusehen hat und wie es sich verhalten soll. Es handelt sich bei einer Klasse also um eine reine Softwarekonstruktion, die Eigenschaften, Methoden und Ereignisse eines Objekts definiert, ohne das Objekt zu erzeugen.

HINWEIS: Oft wird anstatt des Begriffs "Klasse" mit völlig gleichwertiger Bedeutung auch "Objekttyp" verwendet.

Instanz

Man erhält erst dann ein konkretes Objekt, wenn man eine *Instanz* einer Klasse bildet. Es lassen sich viele Objekte mit einer einzigen Klassendefinition erzeugen.

BEISPIEL 3.1: Instanz

Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und denselben Konstruktionsvorschriften für die Klasse *Auto* gebaut. Diesen Vorgang könnte man auch als Bildung von Instanzen der Klasse *Auto* bezeichnen. Während die Klasse lediglich die Eigenschaft *Farbe* definiert, wird der konkrete Wert (rot, blau, grün ...) erst beim Erzeugen des Objekts (der Instanz) zugewiesen.

Kapselung

Klassen realisieren das Prinzip der *Kapselung* von Objekten, das es ermöglicht, die Implementierung der Klasse (der Code im Inneren) von deren Schnittstelle bzw. Interface (die öffentlichen Eigenschaften, Methoden und Ereignisse) sauber zu trennen. Durch das Verbergen der inneren Struktur werden die internen Daten und einige verborgene Methoden geschützt, sind also von außen nicht zugänglich. Die Manipulation des Objekts kann lediglich über streng definierte, über die Schnittstelle zur Verfügung gestellte öffentliche Methoden erfolgen.

Wiederverwendbarkeit

Klassen ermöglichen die *Wiederverwendbarkeit* von Code. Nachdem eine Klasse geschrieben wurde, können Sie diese an verschiedenen Stellen innerhalb einer Applikation verwenden. Klassen reduzieren somit den redundanten Code einer Anwendung, sie erleichtern außerdem die Wartung des Codes.

Vererbung

Echte Vererbung (*Implementierungsvererbung*) ermöglicht es Klassen zu definieren, die von anderen Klassen abgeleitet werden, wobei nicht nur die Schnittstelle, sondern auch der dahinter liegende Code (die Implementierung) vom Nachkommen übernommen wird.

Da es nun möglich ist, die Implementierung einer Klasse für weitere Klassen als Grundlage zu verwenden, kann man Unterklassen bilden, die alle Eigenschaften und Methoden ihrer Oberklasse erben. Diese Unterklassen können zu den geerbten Eigenschaften neue hinzufügen oder Eigenschaften der Oberklasse verstecken, indem sie diese überschreiben.

Wird von einer solchen Unterklasse ein Objekt erzeugt (also eine Instanz der Unterklasse gebildet), dann dient für dieses Objekt sowohl die Ober- als auch die Unterklasse als "Bauplan".

C# unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse (siehe Abschnitt 3.6.2).

Polymorphie

OOP macht es möglich, ein und dieselbe Methode für ganz verschiedene Objekte zu verwenden, man nennt dies dann *Polymorphie* (Vielgestaltigkeit). Jedes dieser Objekte kann die Ausführung unterschiedlich realisieren. Für das aufrufende Objekt bleibt der Vorgang trotzdem derselbe.

BEISPIEL 3.2: Polymorphie

Die Methode *Beschleunigen* ist in einer *Fahrzeug*-Klasse definiert, welche an die Unterklassen *Auto* und *Fahrrad* vererbt. Es ist klar, dass diese Methoden in beiden Unterklassen überschrieben, d.h. völlig unterschiedlich implementiert werden müssen.

Als Polymorphie, die aufs Engste mit der Vererbung verknüpft ist, kann man also die Fähigkeit von Unterklassen bezeichnen, Eigenschaften und Methoden mit dem gleichen Namen, aber mit unterschiedlichen Implementierungen aufzurufen (siehe Abschnitt 3.6.6).

3.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern

Um die Klasse bzw. ihre Mitglieder (Member, Elemente) gezielt zu verbergen oder offen zu legen, sollten Sie von den Zugriffsmodifizierern Gebrauch machen, die den Gültigkeitsbereich (bzw. die *Sichtbarkeit*) einschränken.

Klassen

Die folgende Tabelle zeigt die möglichen Einschränkungen bei der Sichtbarkeit von Klassen:

Modifizierer	Sichtbarkeit
public	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
internal	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
private	Nur innerhalb einer anderen Klasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden (siehe 3.3 2).

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
public	Unbeschränkt.
protected	Innerhalb der Klasse und der daraus abgeleiteten Klassen.
internal	Innerhalb des aktuellen Projekts.
internal protected	Innerhalb des aktuellen Projekts oder der abgeleiteten Klassen.
private	Nur innerhalb der Klasse.

Die Schlüsselwörter *private* und *public* definieren immer die beiden Extreme des Zugriffs. Betrachtet man die jeweiligen Klassen als allein stehend, so reichen diese beiden Zugriffsarten völlig aus. Mit solchen, quasi isolierten, Klassen lassen sich allerdings keine komplexeren Probleme lösen.

Um einzelne Klassen miteinander zu verbinden, verwenden Sie den mächtigen Mechanismus der Vererbung. In diesem Zusammenhang gewinnt die *protected*-Deklaration wie folgt an Bedeutung:

- Da eine abgeleitete Klasse auf die *protected*-Member zugreifen kann, sind diese Member für die abgeleitete Klasse quasi *public*.
- Ist eine Klasse nicht von einer anderen abgeleitet, kann sie nicht auf deren *protected*-Member zugreifen, da diese dann quasi *private* sind.

Mehr zu diesem Thema finden Sie im Abschnitt 3.6 (Vererbung).

3.1.4 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel 2 mit den Strukturen (*struct*, siehe Abschnitt 2.6.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen[1]. Auch im Aufbau von Methoden sollte er sich auskennen (Abschnitt 2.7).

Im Unterschied zu einer Struktur (Schlüsselwort *struct*) wird eine Klasse mit dem Schlüsselwort *class* deklariert. Die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer class Bezeichner
                                {
                                  // ... Felder
                                  // ... Konstruktoren
                                  // ... Eigenschaften
                                  // ... Methoden
                                  // ... Ereignisse
                                }
```

[1] Der wesentliche Unterschied ist der, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind.

Zur Bedeutung der (Zugriffs-)*Modifizierer* wird auf obige Tabellen verwiesen.

Im Klassenkörper haben es wir es mit "Klassenmitgliedern" (Member) wie Feldern, Konstruktoren, Eigenschaften, Methoden und Ereignissen zu tun, auf die wir noch detailliert zu sprechen kommen werden.

Die Definition der Klassenmitglieder bezeichnet man auch als *Implementation* der Klasse.

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

C#

```
public class CKunde
{
   private string _anrede;          //     Feld
   private string _name;            //     dto.

 public CKunde(string anr, string nam)        // Konstruktor
 {
   _anrede = anr;
   _name = nam;
 }

 public string name             // Eigenschaft
   {
        get {return(_name); }
        set {_name = value; }
   }

   public string adresse()               // Methode
   {
        string s = _anrede + " " + _name;
        return(s);
   }
 }
```

Unsere Klasse verfügt damit über zwei Felder, eine Eigenschaft und eine Methode. Da die beiden Felder mit dem *private*-Modifizierer deklariert wurden, sind sie von außen nicht sichtbar.

3.1.5 Das Erzeugen eines Objekts

Existiert eine Klasse, so steht dem Erzeugen von Objektvariablen nichts mehr im Weg. Eine Objektvariable ist ein Verweistyp, sie enthält also nicht das Objekt selbst, sondern stellt lediglich einen Zeiger (Adresse) auf den Speicherbereich des Objekts bereit. Es können sich also durchaus mehrere Objektvariablen auf ein und dasselbe Objekt beziehen. Wenn eine Objektvariable den Wert *null* enthält, bedeutet das, dass sie momentan "ins Leere" zeigt, also kein Objekt referenziert.

Unter der Voraussetzung, dass eine gültige Klasse existiert, verläuft der Lebenszyklus eines Objekts in Ihrem Programm in folgenden Etappen:

- Referenzierung (eine Objektvariable wird deklariert, sie verweist momentan noch auf *null*)
- Instanziierung (die Objektvariable zeigt jetzt auf einen konkreten Speicherplatzbereich)
- Initialisierung (die Datenfelder der Objektvariablen werden mit Anfangswerten gefüllt)
- Arbeiten mit dem Objekt (es wird auf Eigenschaften und Methoden des Objekts zugegriffen, Ereignisse werden ausgelöst)
- Zerstören des Objekts (das Objekt wird dereferenziert, der belegte Speicherplatz wird wieder freigegeben)

Werfen wir nun einen genaueren Blick auf die einzelnen Etappen.

Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

In *Variante 1* wird pro Schritt eine Anweisung verwendet:

SYNTAX:
```
Modifizierer Klasse Object;
Object = new Klasse(Parameter);
```

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

C#
```
private CKunde kunde1;                    // Referenzieren
kunde1 = new CKunde1();                   // Erzeugen
```

In *Variante2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

SYNTAX:
```
Klasse Object = new Klasse();
```

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

C#
```
private CKunde kunde1 = new CKunde();
```

Dem Klassenbezeichner (*Klasse*) müsste genauer genommen noch der Name der Klassenbibliothek (bzw. Name des Projekts) vorangestellt werden, doch dies wird unter Visual Studio nicht erforderlich sein, da der entsprechende Namensraum (*Namespace*) bereits automatisch eingebunden wurde (*using*-Anweisung).

Obwohl die Kurzform sehr eindrucksvoll ist, können Sie hier keine Fehlerbehandlung (*try...catch*-Block) durchführen. Diese Einschränkung macht diese Art von Deklaration weniger nützlich.

Empfehlenswert ist also fast immer das getrennte Deklarieren und Erzeugen[1].

[1] Aus Platzgründen halten sich die Autoren leider nicht immer an diese Empfehlung.

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

C#
```
private CKunde kunde1;
try
{
     kunde1 = new CKunde();
}
catch(Exception ex)
{
 MessageBox.Show(ex.Message);
}
```

Klassische Initialisierung

Statt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

C#
```
CKunde kunde1 = new CKunde();
kunde1.anrede = "Frau"; kunde1.name = "Müller";
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

C#
```
CKunde kunde1 = new CKunde("Frau", "Müller");
```

Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.5.1.

Objekt-Initialisierer

Ab C# 3.0 wurden – vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Abschnitt 2.2) – so genannte *Objektinitialisierer* eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden. Das Initialisieren geschieht über geschweifte Klammern, in denen die einzelnen Felder/Eigenschaften des Objekts mit Werten belegt werden.

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

C#
```
public class CPerson
{
      public string Name;
      public string Strasse;
      public int PLZ;
      public string Ort;
}
```

BEISPIEL 3.9: Gegeben ist eine Klasse *CPerson*:

Das Erzeugen und Initialisieren einer Instanz von *CPerson* bedarf keines Konstruktors:

```
CPerson person1 = new CPerson { Name = "Müller", Strasse = "Am Waldesrand 7", PLZ = 12345,
                                Ort = "Musterhausen" };
```

Arbeiten mit dem Objekt

Wie Sie bereits wissen, erfolgt der Zugriff auf Eigenschaften und Methoden eines Objekts, indem der Name des Objekts mit einem Punkt (.) vom Namen der Eigenschaft/Methode getrennt wird.

SYNTAX: *Objekt.Eigenschaft|Methode()*

BEISPIEL 3.10: Die Eigenschaft *guthaben* des Objekts *kunde1* wird zugewiesen und die Methode *adresse* aufgerufen

C#
```
kunde1.guthaben = 10;
label1.Text = kunde1.adresse();
```

Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *null* setzen.

BEISPIEL 3.11: Der *kunde1* wird in die ewigen Jagdgründe befördert

C#
```
kunde1 = null;
```

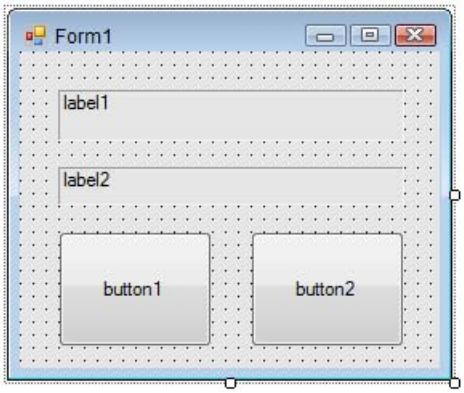
Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird (siehe Abschnitt 3.5.2).

3.1.6 Einführungsbeispiel

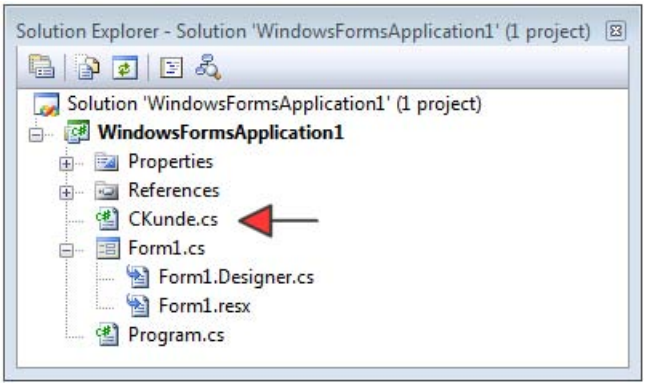
Raus aus dem muffigen Hörsaal, lasst uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

Vorbereitungen

- Öffnen Sie ein neues Projekt (z.B. mit dem Namen "Kunden") als Windows Forms-Anwendung.
- Auf das Startformular (*Form1*) platzieren Sie zwei *Label*s und zwei *Button*s.



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.cs* ein und klicken "Hinzufügen". Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



Sie müssen eine Klasse nicht unbedingt in einem eigenen Klassenmodul definieren, Sie könnten die Klasse z.B. auch zum bereits vorhandenen Code des Formulars (*Form1.cs*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

Klasse definieren

Im Code-Fenster *CKunde.cs* ist bereits der Rahmencode für unsere Klasse vorbereitet:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace WindowsFormsApplication1
{
    class CKunde
    {
    }
}
```

Tragen Sie dann in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
public class CKunde
{
    private const char LF = (char) 10;  //  private Konstante (Zeilenumbruch)

    public string anrede;          //    öffentliches Feld
    public string name;            //      dto.
    public int plz;                //      dto.
    public string ort;             //      dto.
    public bool stammkunde;        //      dto.
    public decimal guthaben;       //      dto.

    public string adresse()                 //    öffentliche Methode
    {
      string s = anrede + " " + name + LF + plz.ToString() + " " + ort;
      return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
        if (stammkunde) guthaben += betrag;
    }
}
```

Bemerkungen

- Die Klasse verfügt über sechs "einfache" Eigenschaften, und zwar sind das alle als *public* deklarierten Variablen, die man auch als "öffentliche Felder" bezeichnet. Die Betonung liegt hier auf "einfach", da wir später noch lernen werden, wie man "richtige" Eigenschaften programmiert.
- Weiterhin verfügt die Klasse über zwei *Methoden*. Die *string*-Methode *adresse()* liefert einen Rückgabewert, nämlich die komplette Anschrift.
- Die *void*-Methode *addGuthaben* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *guthaben*-Felds bei jedem Aufruf um 50 €.
- Die private Konstante *LF* wird von der Methode *adresse()* für das Einfügen des Zeilenumbruchs benötigt.

Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Auf Klassenebene deklarieren Sie eine Objektvariable *kunde1*:

```
private CKunde kunde1;                // Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen das *Click*-Ereignis wie folgt:

```
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();          // Objekt erzeugen

    // Objektfelder initialisieren:
    kunde1.anrede = "Herr";
    kunde1.name = "Müller";
    kunde1.plz = 12345;
    kunde1.ort = "Berlin";
    kunde1.stammkunde = true;
}
```

Objekt verwenden

Hinterlegen Sie nun den rechten Button mit der Beschriftung "Methoden und Eigenschaften verwenden" wie folgt:

```
private void button2_Click(object sender, System.EventArgs e)
{
    label1.Text = kunde1.adresse();  // erste Methode aufrufen
    kunde1.addGuthaben(50M);         // zweite Methode aufrufen
    label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");     // Eigenschaft lesen
}
```

Unterstützung durch die IntelliSense

Sie haben beim Eintippen des Quelltextes (insbesondere im Code-Fenster von *Form1*) bereits gemerkt, dass Sie durch die IntelliSense von Visual Studio eifrigst unterstützt werden.

Die IntelliSense weist Sie z.B. auf die verfügbaren Klassenmitglieder (Eigenschaften und Methoden) hin und ergänzt den Quellcode automatisch, wenn Sie auf den gewünschten Eintrag doppelklicken.

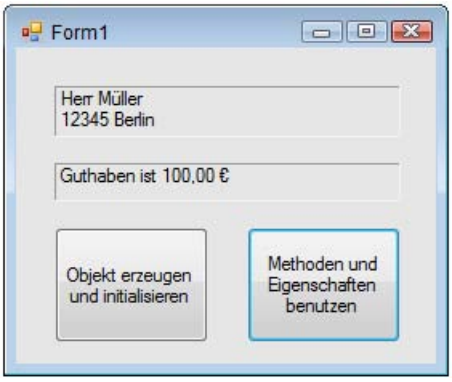
```
// Objekt erzeugen und initialisieren:
private void button1_Click(object sender, System.EventArgs e)
{
    kunde1 = new CKunde();
    kunde1.
    kund                                    er";
    kund
    kund                                    n";
    kund                                     true;
}
// Method                                  aften benutzen:
private                                     ick(object sender, System.EventArgs e)
{
    labe                                    1.Adresse();     // erste Methode aufrufen
    kund                                    50M);          // zweite Methode aufrufen
```

addGuthaben
Adresse
anrede
Equals
GetHashCode
GetType
guthaben
name
ort
plz

Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *public*-Modifizierer vergessen wurde.

Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes C#-Objekt vom Stapel lassen können. Unmittelbar nach Programmstart betätigen Sie den linken Button und danach den rechten. Durch mehrmaliges Klicken auf den zweiten Button wird sich das Guthaben des Kunden Müller in 50-€-Schritten erhöhen.



Falls Sie zu voreilig gewesen sind und unmittelbar nach Programmstart den zweiten statt den ersten Button gedrückt haben, stürzt Ihnen das Programm mit der Laufzeit-Fehlermeldung "Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt." ab.

Bemerkungen

Unsere Klasse funktioniert nach außen hin zwar ohne erkennbare Mängel, ist hinsichtlich ihrer inneren Konstruktion aber keinesfalls als optimal zu bezeichnen. Wir haben deshalb keinerlei Grund, uns zufrieden zurückzulehnen, denn das uns unter C# zur Verfügung stehende OOP-Instrumentarium wurde von uns bei weitem noch nicht ausgeschöpft.

- Beispielsweise haben wir nur "einfache" Eigenschaften, nämlich *public*-Felder verwendet, was eigentlich eine schwere Sünde in den Augen der OOP-Puristen ist (siehe Abschnitt 3.2.1).
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht, siehe Abschnitt 3.5.1).
- Außerdem wird eine Klasse erst dann so richtig effektiv, wenn wir davon nicht nur eine, sondern mehrere Instanzen (sprich Objekte) ableiten. Diese wiederum kann man ziemlich elegant in so genannten Auflistungen (Collections) verwalten (siehe Kapitel 5).

Doch zur Beseitigung dieser und anderer Unzulänglichkeiten kommen wir erst später. Ein weiteres Problem, was uns unter den Nägeln brennt, duldet keinen weiteren Aufschub und wir wollen es deshalb gleich im folgenden Abschnitt behandeln.

3.2 Eigenschaften

Eigenschaften bestimmen die statischen Attribute eines Objekts, sie leiten sich von dessen *Zustand* ab, wie er in den Zustandsvariablen (Objektfeldern) gespeichert ist. Im Unterschied zu den Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden, sind die den Eigenschaften zugewiesenen Werte für alle Objekte einer Klasse meist unterschiedlich.

3.2.1 Eigenschaften mit Zugriffsmethoden kapseln

Von den im Objekt enthaltenen Feldern sind die *public*-Felder als "einfache" Eigenschaften zu betrachten.

In unserem Beispiel hatten wir für die Klasse *CKunde* solche "einfachen" Eigenschaften als *public*-Variable deklariert. Das allerdings ist nicht die "feine Art" der objektorientierten Programmierung, denn das Veröffentlichen von Feldern widerspricht dem hochgelobten Prinzip der Kapselung und erlaubt keinerlei Zugriffskontrolle wie z.B. Wertebereichsüberprüfung oder die Vergabe von Lese- und Schreibrechten.

Idealerweise sind deshalb in einem Objekt nur private Felder enthalten, und der Zugriff auf diese wird durch Accessoren (Zugriffsmethoden) gesteuert.

In diesem Sinn ist eine *Eigenschaft* gewissermaßen ein Mittelding zwischen Feld und Methode. Sie verwenden die Eigenschaft wie ein öffentliches Feld. Vom Compiler aber wird der Feldzugriff in den Aufruf von Accessoren – das sind spezielle Zugriffsmethoden auf private Felder – übersetzt. Doch schauen wir uns das Ganze lieber in der Praxis an.

Deklarieren von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *set*- und für den Schreibzugriff eine g*et*-Zugriffsmethode. Während die *get*-Methode ihren Rückgabewert über *return* liefert, erhält die *set*-Methode den zu schreibenden Wert über *value*.

SYNTAX:
```
Modifizierer Datentyp Eigenschaftsname
{
    get
    {
        // hier Lesezugriff auf priv. Felder implementieren
        return(privatesFeld);
    }
    set
    {
        // hier Schreibzugriff auf priv. Felder implementieren
        privatesFeld = value;
    }
}
```

Wir wollen nun unser Beispiel mit "echten" Eigenschaften ausstatten. Dazu werden zunächst die public-Felder in *private* verwandelt und durch Voranstellen von "_" umbenannt, um Namenskonflikte mit den gleichnamigen Eigenschafts-Deklarationen zu vermeiden.

Der Schreibzugriff auf die Eigenschaft *anrede* wird so kontrolliert, dass nur die Werte "Herr" oder "Frau" zulässig sind.

```
public class CKunde
{
    private string _anrede;          //    privates Feld
    private string _name;            //      dto.
    ...

    public string anrede
    {
        get {return(_anrede); }
        set
        {
            if (value == "Herr" || value == "Frau") _anrede = value;
            else MessageBox.Show("Die Anrede '" + value + "' ist nicht zulässig!");
        }
    }

    public string name
    {
        get {return(_name); }
        set {_name = value; }
    }
```

```
    ...
  }
}
```

Zugriff

Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *get*- bzw. *set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

C#

```
kunde1 = new CKunde();
kunde1.anrede = "Mister";      // Fehler!
```

Ergebnis

Bemerkung

- Beim Schreiben des Quellcodes in der Entwicklungsumgebung Visual Studio merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die IntelliSense zeigt dafür unterschiedliche Symbole[1].
- In unserem Beispiel verhält sich nur die Eigenschaft *anrede* "intelligent", d h., sie unterliegt einer Zugriffskontrolle. Bei den übrigen Eigenschaften erfolgt lediglich eine 1:1-Zuordnung zu den privaten Feldern. Hier sollte man nicht "päpstlicher als der Papst" sein und es bei den ursprünglichen *public*-Feldern belassen. Wir aber haben diesen (eigentlich sinnlosen) Aufwand nur wegen des Lerneffekts betrieben.

3.2.2 Berechnete Eigenschaften

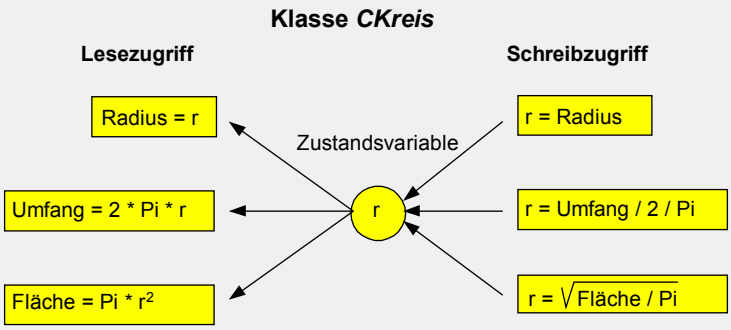
Mit Zugriffsmethoden lässt sich weit mehr anstellen, als nur den Zugriff auf private Felder der Klasse zu kontrollieren. So können z.B. innerhalb der Methode komplexe Berechnungen mit den Feldern (die man auch *Zustandsvariablen* nennt) und den übergebenen Parametern ausgeführt werden.

[1] Probieren Sie das bitte selbst aus!

BEISPIEL 3.13: Berechnete Eigenschaften

C#

Eine Klasse *CKreis* hat die Eigenschaften *radius*, *umfang* und *fläche*. In der einzigen Zustandsvariablen *r* braucht aber nur der Radius abgespeichert zu werden, da sich die übrigen Eigenschaften aus *r* berechnen lassen (*get* = Lesezugriff) bzw. umgekehrt (*set* = Schreibzugriff).




```
public class CKreis
{
    private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Eigenschaft *radius*:

```
    public string radius
    {
       get {return (r.ToString("#,#0.00")); }
       set
       {
           if (value != "")  r = Convert.ToDouble(value);
           else r = 0;
       }
    }
```

Die Eigenschaft *umfang*:

```
    public string umfang
    {
        get {return (2 * Math.PI * r).ToString("#,#0.00"); }
        set
        {
            if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
            else r = 0;
        }
    }
```

Die Eigenschaft *fläche*:

```
    public string fläche
    {
```

BEISPIEL 3.13: Berechnete Eigenschaften

```
        get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
        set
        {
            if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
            else r = 0;
        }
    }
}
```

Das komplette Programm finden Sie im Praxisbeispiel

► 3.9.1 Eigenschaften sinnvoll kapseln

3.2.3 Lese-/Schreibschutz

Es kommt häufig vor, dass bestimmte Felder bzw. Eigenschaften nur gelesen oder nur geschrieben werden dürfen.

Für Felder kann man einen Schreibschutz einfach durch Voranstellen des *readonly*-Modifizierers realisieren.

BEISPIEL 3.14: Ein schreibgeschütztes öffentliches Feld wird deklariert und initialisiert.

C#
```
public readonly double mwst = 0.19;
```

HINWEIS: Außer beim Deklarieren kann man ein ReadOnly-Feld auch in einem Konstruktor initialisieren! Genau dadurch unterscheidet sich das *readonly*- vom *const*-Schlüsselwort, denn *readonly*-Felder können – abhängig vom verwendeten Konstruktor – über unterschiedliche Werte verfügen.

Verwendet man statt öffentlicher Felder "richtige" Eigenschaften, so ist für eine Zugriffsbeschränkung keinerlei zusätzlicher Aufwand erforderlich – im Gegenteil:

HINWEIS: Um eine Eigenschaft allein für den Lese- bzw. Schreibzugriff zu deklarieren, lässt man einfach die *get*- bzw. die *set*-Zugriffsmethode weg.

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

C#
Das klingt sicher logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
public decimal guthaben
{
    get {return(_guthaben); }
}
```

BEISPIEL 3.15: In der *CKunde*-Klasse soll das Guthaben für den Schreibzugriff gesperrt werden

Ergebnis

In der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:

```
kunde1.stammkunde = true;
kunde1.guthaben = 10M;
```

Einer Eigenschaft oder einem Indexer 'Kunden.CKunde.guthaben' kann nicht zugewiesen werden -- sie sind schreibgeschützt

3.2.4 Property-Accessoren

Es möglich, den Zugriff auf *get*- oder *set*- Accessoren von Eigenschaften zu beschränken. Meist ist dies nur für den *set*-Accessor sinnvoll, während der *get*-Accessor in der Regel öffentlich bleibt.

BEISPIEL 3.16: Property-Accessoren

C#

Eine Eigenschaft mit *get*- und *set*-Accessoren. Der *get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *set*-Accessor nur einen *protected*-Zugriff erlaubt.

```
public string KontoNummer
{
    get
    {
        return _knr;
    }
    protected set
    {
        _knr = value;
    }
}
```

3.2.5 Statische Felder/Eigenschaften

Mitunter gibt es Felder bzw. Eigenschaften, deren Werte für alle aus der Klasse instanziierten Objekte identisch sind und die deshalb nur einmal in der Klasse gespeichert zu werden brauchen.

HINWEIS: Statische Felder/Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *static* deklariert.

Statische Eigenschaften bzw. öffentliche Felder können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden müsste! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.17: Die Klasse *CKunde* soll zusätzlich ein öffentliches Feld (bzw. eine "einfache" Eigenschaft) *rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

C#
```
public class CKunde
{
    ...
    public static double rabatt;
    ...
}
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.18: Allen Kunden wird ein Rabatt von 15% zugewiesen.

C#
```
CKunde.rabatt = 0.15;
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre geliebten globalen Variablen zu verzichten, mit denen sie bequem Werte zwischen verschiedenen Programmmodulen austauschen konnten. Genau hier bieten sich statische Eigenschaften bzw. öffentliche statische Felder an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

C#
```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst);}
        set { _mwst = value; }
    }
    ...
```

Hier könnten z.B. auch nichtstatische Klassenmitglieder eingefügt werden, die natürlich auch auf das statische Feld *_mwst* zugreifen dürfen.

```
}
```

Sowohl von *Form1* als auch von *Form2* aus kann direkt auf die statische Eigenschaft *MWst* zugegriffen werden, eine Instanz von *CAllerlei* braucht dazu nicht erzeugt zu werden:

Zuweisen der Mehrwertsteuer in *Form1*:

```
public partial class Form1 : Form
{
    ...
```

BEISPIEL 3.19: Die Klassen *Form1* und *Form2* greifen für allgemeine Berechnungen auf eine statische Eigenschaft *MWSt* der Klasse *CAllerlei* zu.

```
        CAllerlei.MWSt = 0.19;
    ...
}
```

Anzeige der Mehrwertsteuer in *Form2*:

```
public partial class Form2 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
        textBox1.Text = CAllerlei.MWSt.ToString();
    }
}
```

Es versteht sich, dass innerhalb der *get*- bzw. *set*-Accessoren statischer Eigenschaften nur auf statische Klassenmitglieder (Felder) zugegriffen werden kann. Gleiches gilt übrigens auch für statische Methoden (siehe 3.3.3).

Konstante Felder

Obwohl ein Feld nicht als *static const* deklariert werden kann, verhält sich ein *const*-Feld im Wesentlichen statisch. Deshalb kann auf *const*-Felder mit der gleichen Notation wie bei *static*-Feldern zugegriffen werden (eine mit *new* erzeugte Objektinstanz ist nicht erforderlich).

BEISPIEL 3.20: Deklaration einer öffentlichen Konstanten und Zugriff

C#

```
public class CAllerlei
{
    public const int anzahl = 50;
    ...
}
```

Der Zugriff ist direkt möglich:

```
public partial class Form1 : Form
{
    ...
    n = CAllerlei.anzahl;
    ...
}
```

Worin unterscheiden sich denn dann statische und konstante Felder? Der wesentliche Unterschied ist der, dass man den Wert statischer Felder beim Erzeugen eines Objekts (durch Aufruf unterschiedlicher Konstruktoren) oder zur Laufzeit (durch Methodenaufrufe) ändern kann, bei konstanten Feldern geht das natürlich nicht.

3.2.6 Einfache Eigenschaften automatisch implementieren

Wie bereits im Abschnitt 3.2.1 erwähnt, gehört es zum schlechten Programmierstil, wenn *public*-Variablen quasi die Rolle von einfachen Eigenschaften übernehmen. Getreu der Devise "Hauptsache es funktioniert" ist es aber für den schreibfaulen Programmierer oft der bequemere Weg, erspart er sich damit doch viele Zeilen stupiden Codes. Automatisch implementierte Eigenschaften befreien den Programmierer aus diesem Zwiespalt, sie benötigen in der Regel auch nur eine einzige Codezeile, trotzdem erfolgt im Hintergrund eine exakte Implementierung mit *get*- und *set*-Zugriffsmethoden.

BEISPIEL 3.21: Drei Varianten für eine einfache Eigenschaft *Nachname.*

C#

1. Der kurze, aber schlechte Programmierstil:

```
public class CKunde
{
    public string Nachname;
    ...
}
```

2. Die exakte, aber umständliche Schreibweise:

```
public class CKunde
{
    private string _nachname;
    public string Nachname
    {
        get {return (_nachname);}
        set {_nachname = value;}
    }
    ...
}
```

3. Die Eigenschaft wird automatisch und sauber implementiert:

```
public class CKunde
{
    public string Nachname { get; set; }
    ...
}
```

Wie Sie sehen, ist nur eine einzige Codezeile erforderlich: Der Compiler erstellt hier ein privates, anonymes dahinter liegendes Feld (analog zu *_nachname* bei der ersten Variante), das nur durch die *get*- und *set*-Accessoren aufgerufen werden kann.

HINWEIS: Soll die Eigenschaft schreibgeschützt sein, so legen Sie einfach einen privaten *set*-Accessor fest.

BEISPIEL 3.22: Die automatisch implementierte Eigenschaft *Nachname* ist schreibgeschützt.

C#
```
public string Nachname { get; private set; }
```

HINWEIS: Es sei hier nochmals betont, dass sich nur einfache Eigenschaften automatisch implementieren lassen, keine berechneten Eigenschaften!

3.3 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.3.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben wir gelernt, wie man Methoden programmiert. Jetzt wollen wir noch etwas nachhaken und den Fokus auf die Methoden richten, die in unseren selbst programmierten Klassen zum Einsatz kommen.

Genau wie das bei "richtigen" Eigenschaften der Fall ist, arbeiten in einer sauber programmierten Klasse alle Methoden ausschließlich mit privaten Feldern (Zustandsvariablen) zusammen.

HINWEIS: Wenn Sie eine Methode als *private* deklarieren, ist sie nur innerhalb der Klasse sichtbar, und es handelt sich um keine Methode im eigentlichen Sinn der OOP, sondern eher um eine Funktion/Prozedur im herkömmlichen Sinn.

BEISPIEL 3.23: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

C#
```
public class CKunde
{
    private string _anrede;          //    privates Feld
    private string _name;            //      dto.
    private int _plz;                //      dto.
    private string _ort;             //      dto.
    private bool _stammkunde;        //      dto.
    private decimal _guthaben;       //      dto.

    public string adresse()          //    öffentliche Methode
    {
        string s = _anrede + " " + _name  + _plz.ToString() + " " + _ort;
        return(s);
    }

    public void addGuthaben(decimal betrag)   // öffentliche Methode
    {
```

BEISPIEL 3.23: Die Methoden *adresse* und *addGuthaben* arbeiten mit sechs privaten Feldern zusammen

```
        if (stammkunde) _guthaben += betrag;
    }
}
```

Deklaration und Aufruf:

```
CKunde kunde1 = new CKunde();
...
label1.Text = kunde1.adresse();   // erste Methode aufrufen
kunde1.addGuthaben(50M);          // zweite Methode aufrufen
```

3.3.2 Überladene Methoden

Obwohl wir bereits in Abschnitt 2.7.5 ganz allgemein auf dieses Thema eingegangen sind, soll es hier nochmals im OOP-Kontext diskutiert werden.

Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

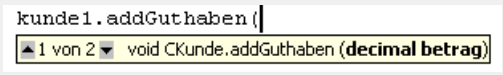
BEISPIEL 3.24: Überladene Methoden

C#

Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

```
public void addGuthaben(decimal betrag)              // erste Überladung
{
  if (stammkunde) _guthaben += betrag;
}

public void addGuthaben(decimal brutto, decimal mwst)   // zweite Überladung
{
  _guthaben += Convert.ToDecimal(brutto/(1 + mwst));
}
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.3.3 Statische Methoden

Genauso wie die unter 3.2.5 erläuterten *statischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden eignen sich z.B. gut für diverse Formelsammlungen (ähnlich *Math*-Klassenbibliothek). Auch können Sie damit auf private statische Klassenmitglieder zugreifen.

HINWEIS: Der Einsatz statischer Methoden für relativ einfache Aufgaben ist bequemer und ressourcenschonender als das Arbeiten mit Objekten, die Sie jedes Mal extra instanziieren müssten.

BEISPIEL 3.25: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

C#
```
public class CMeineFormeln
{
    public static double kreisUmfang(double radius)
    {
       return (2 * Math.PI * radius);
    }

    public static double kugelVolumen(double radius)
    {
       return ( 4 / 3.0 * Math.PI * Math.Pow(radius, 3));
    }

    public static decimal netto(decimal brutto, double mwst)
    {
        return(brutto/ Convert.ToDecimal(1 + mwst));
    }

   // .... weitere Methoden
}
```

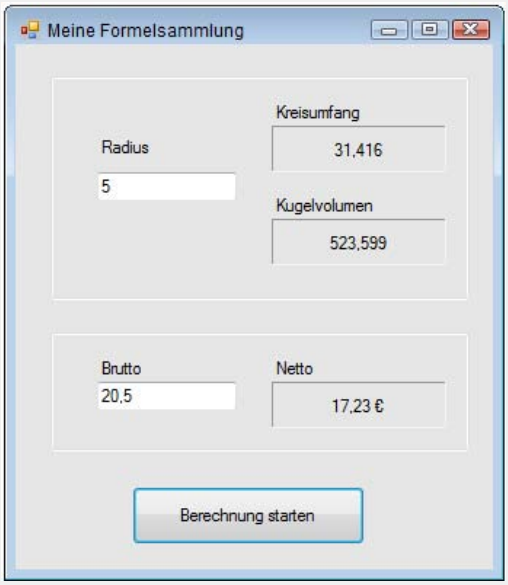
Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

BEISPIEL 3.26: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

C#
```
private void button1_Click(object sender, EventArgs e)
{
     double r = Convert.ToDouble(textBox1.Text);              // Kreisradius konvertieren

     label1.Text = CMeineFormeln.kreisUmfang(r).ToString("0.000");
     label2.Text = CMeineFormeln.kugelVolumen(r).ToString("0.000");

     decimal b = Convert.ToDecimal(textBox2.Text);            // Brutto konvertieren
     label3.Text = CMeineFormeln.netto(b, 0.19).ToString("C");
}
```

BEISPIEL 3.26: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

Ergebnis



Bemerkungen

- Sie können mit *static* auch ein Feld deklarieren, das von allen Instanzen der Klasse gemeinsam genutzt werden kann und nicht für jedes Objekt extra zugewiesen werden muss. Zum Initialisieren kann ein so genannter *statischer Konstruktor* Verwendung finden (siehe 3.5.1).
- Eine Klasse mit ausschließlich statischen Mitgliedern kann mit dem Schlüsselwort *static* deklariert werden, siehe 3.7.4 oder Praxisbeispiel

  ► 3.9.2 Eine statische Klasse anwenden.

3.4 Ereignisse

Nachdem wir uns den Eigenschaften und Methoden von Objekten ausführlich gewidmet haben, wollen wir die Dritten im Bunde, die Ereignisse, nicht vergessen. Wie Sie bereits wissen, werden Ereignisse unter bestimmten Bedingungen vom Objekt ausgelöst und können dann in einer Ereignisbehandlungsroutine abgefangen und ausgewertet werden.

Allerdings bieten bei weitem nicht alle Klassen Ereignisse an, denn diese werden nur benötigt, wenn auf bestimmte Änderungen eines Objekts reagiert werden soll.

Nachdem wir mit dem Deklarieren von Eigenschaften und Methoden überhaupt keine Probleme hatten, hört aber bei Ereignissen der Spaß auf.

HINWEIS: Eine ausführliche Einführung in das Ereignismodell erhalten Sie im Kapitel 27 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.4.1 Ereignis hinzufügen

Um einer Klasse ein Ereignis hinzuzufügen, sind drei Schritte erforderlich:

1. Die Deklaration des Ereignistyps (*delegate*-Schlüsselwort)
2. Die Instanziierung des Ereignisses (*event*-Schlüsselwort)
3. Das Auslösen des Ereignisses (innerhalb einer Methode oder Eigenschaft)

Um einer heillosen Verwirrung vorzubeugen, machen wir es diesmal umgekehrt und beginnen gleich mit einem Beispiel, ehe wir später die Syntax und weitere Einzelheiten erklären.

BEISPIEL 3.27: Ereignis-Delegate hinzufügen

C#

In unserer *CKunde*-Klasse wird ein Ereignis-Delegate mit dem Namen *GuthabenLeer* deklariert, davon wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert. Dieses Ereignis "feuert" innerhalb der Methode *addGuthaben* genau dann, wenn das Guthaben den Wert von 10 € unterschreitet.

```
public class CKunde
{
```

Um den Code für den Benutzer der Klasse etwas zu vereinfachen, werden den privaten Feldern Standardwerte zugewiesen[1]:

```
    private bool _stammkunde = true;        //      initialisiertes Feld
    private decimal _guthaben = 100M;       //      dto.
```

1.Schritt: den Ereignistyp definieren:

```
    public delegate void GuthabenLeer(object sender, string e);
```

2. Schritt: eine Ereignisinstanz *guthabenLeer1* deklarieren:

```
    public event GuthabenLeer guthabenLeer1;
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
    public void addGuthaben(decimal betrag)
    {
        if (stammkunde) _guthaben += betrag;
```

3. Schritt: Ereignis auslösen:

```
        if (_guthaben <= 10)
        {
```

[1] Später werden wir diese Aufgabe dem Konstruktor übertragen.

BEISPIEL 3.27: Ereignis-Delegate hinzufügen

```
            // das Ereignis feuert nur, wenn ...
            if (guthabenLeer1 != null)  // ... mindestens ein Eventhandler angemeldet ist
            {
                string msg = "Das Guthaben beträgt nur noch " +
                             _guthaben.ToString("C") + "!";
                guthabenLeer1(this, msg);
            }
        }
    }
    // ... weitere Implementierungen
}
```

Nun kommen wir zu den sicherlich dringend notwendigen Erklärungen:

Ereignis deklarieren

Wie bereits kurz erwähnt, werden Ereignisse von so genannten Delegaten abgeleitet. Ein Delegate ist ein Ereignistyp, er sieht – bis auf das *delegate*-Schlüsselwort – wie eine Methode aus und verhält sich auch ähnlich.

HINWEIS: Delegaten ermöglichen es, ein Framework von Rückruf- und Benachrichtigungsmethoden für miteinander kooperierende Klassen zu implementieren.

SYNTAX: *Modifizierer* **delegate** *Datentyp delegateName* (*Datentyp Parameter*);

Falls der Delegate keinen Rückgabewert liefert, wird dieser – wie bei einer Methode – als *void* angegeben.

BEISPIEL 3.28: Die Deklaration des Delegaten aus dem Vorgängerbeispiel

C#
```
public delegate void GuthabenLeer(object sender, string e);
```

Ereignis instanziieren

Nachdem Sie mittels Delegaten den Ereignistyp deklariert haben, steht dem Erzeugen einer Delegatinstanz, also eines spezifischen Ereignisses, nichts mehr im Wege. Sie verwenden dazu das *event*-Schlüsselwort.

SYNTAX: *Modifizierer* **event** *delegateName ereignisName*;

Ähnlich wie bei Objekten (diese sind bekanntlich Instanzen einer Klasse) handelt es sich bei einem Ereignis um eine Instanz des Delegaten. Da der Aufbau bereits feststeht, genügen die Angabe des Namens der Ereignisdeklaration (*delegateName*) und der spezielle Name des Ereignisses (*ereignisName*).

BEISPIEL 3.29: Von im Vorgängerbeispiel deklarierten Delegaten wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert.

C#
```
public event GuthabenLeer guthabenLeer1;
```

Es ist durchaus möglich und üblich, auch mehrere Ereignisse vom gleichen Delegaten abzuleiten.

Ereignis auslösen

Ein Ereignis wird immer innerhalb der Klasse ausgelöst, in der es deklariert und erzeugt wurde. Das kann an verschiedenen Stellen innerhalb von Methoden oder Eigenschaften geschehen.

Das Auslösen erfolgt wie ein normaler Methodenaufruf:

SYNTAX: *ereignisName*(*Parameter*);

Die Signatur der Parameter (Reihenfolge und Datentyp) muss der im entsprechenden Delegate festgelegten Parameterliste entsprechen.

HINWEIS: Ereignisse sind Verweistypen und können demzufolge auch auf *null* abgefragt werden.

BEISPIEL 3.30: Ereignis auslösen

C#

Das im Vorgängerbeispiel deklarierte Ereignis wird innerhalb der Methode *addGuthaben* ausgelöst[1]. Vor dem Aufruf wird getestet, ob zumindest ein Eventhandler für dieses Ereignis angemeldet ist (was genau unter "Anmelden" zu verstehen ist, erfahren Sie im nächsten Abschnitt).

```
public void addGuthaben(decimal betrag)
{
    if (stammkunde) _guthaben += betrag;
    if (_guthaben <= 10)
    {
        if (guthabenLeer1 != null)    // mindestens ein Eventhandler angemeldet?
        {
            string msg = "Das Guthaben beträgt nur noch " +
                         _guthaben.ToString("C") + "!";
            guthabenLeer1(this, msg);
         }
    }
}
```

[1] Im Programmiererjargon sagt man auch "Das Ereignis feuert"!

3.4.2 Ereignis verwenden

In der Klasse, in welcher wir mit dem Ereignis arbeiten wollen, sind – zusätzlich zur Erzeugung der Objektvariablen – zwei Schritte durchzuführen:

1. Ereignisbehandlung (Eventhandler) schreiben
2. Eventhandler anmelden

Lassen Sie uns auch hier mit einem Beispiel beginnen.

BEISPIEL 3.31: Ereignisse verwenden

C#

Wir nutzen die im Vorgängerabschnitt definierte Klasse *CKunde*, welche von uns gerade mit dem Ereignis *guthabenLeer1* nachgerüstet wurde.

Auf Klassenebene referenzieren wir zunächst die übliche Objektvariable:

```
private CKunde kunde1;            // Objekt referenzieren
```

1. Schritt: Wir schreiben nun eine Ereignisbehandlung (Eventhandler) für das Ereignis:

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde)o;
    label2.Text = k.nachName + ": " + s;       // Ausgabe einer Warnung
}
```

2. Schritt: Um das Ereignis mit dem Eventhandler zu verbinden, ist eine Anmeldung erforderlich, die wir in den Konstruktorcode von *Form1* einfügen können:

```
public Form1()
{
       InitializeComponent();
       kunde1 = new CKunde();      // Objekt instanziieren

       // Eventhandler anmelden:
       kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
}
```

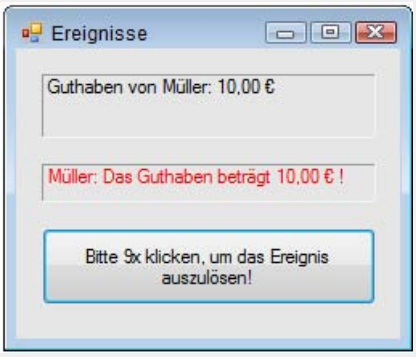
Das Ereignis ist jetzt eingebunden, und einem Funktionstest steht nichts mehr im Weg. Dazu rufen wir wiederholt die Methode *addGuthaben* auf, die das Guthaben jedes Mal um 10 € verringert:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = kunde1.adresse();
    kunde1.addGuthaben(-10M);         // Guthaben verringern
    label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");
}
```

BEISPIEL 3.31: Ereignisse verwenden

Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, wird das Ereignis ausgelöst:



Den kompletten Quellcode entnehmen Sie bitte den Buchbeispielen.

Nun zu den Details.

Ereignisbehandlung schreiben

Die Frage "Was soll passieren, wenn das Ereignis ausgelöst wurde?" wird in einer Ereignisbehandlungsmethode (Eventhandler) beantwortet.

SYNTAX: *Modifizierer Datentyp methodenName (Datentyp Parameter)*

Den Namen der Methode können Sie frei wählen. Den konkreten Namen des Ereignisses finden Sie hier nicht, d.h., eine Ereignisbehandlung lässt sich auch von mehreren Ereignissen gemeinsam verwenden. Lediglich die Methodensignatur (*Datentyp Parameter*) muss der des Delegates entsprechen, nach dessen Muster das Ereignis erzeugt wurde.

BEISPIEL 3.32: Ereignisbehandlung schreiben

C#

Beim Auftreten des Ereignisses wird nicht nur der Parameter *s* angezeigt, in der Titelleiste erscheint zusätzlich noch der Name des Kunden, den wir durch explizite Typkonvertierung aus dem ebenfalls übergebenen *object*-Parameter "herausziehen".

```
private void guthabenKontrolle(object o, string s)
{
    CKunde k = (CKunde) o;                // Typecasting
    label2.Text = k.nachName + ": " + s;  // Warnung ausgeben
}
```

Ereignisbehandlung anmelden

Um dem Compiler mitzuteilen, welcher Eventhandler bei Auftreten des Ereignisses denn nun aufzurufen ist, müssen Sie die gewünschte Ereignisbehandlung beim Objekt (der Klasseninstanz) anmelden.

SYNTAX: *Objekt.ereignisName* += new *delegateName*(*eventHandlerName*);

BEISPIEL 3.33: Dem Objekt *kunde1* wird mitgeteilt, dass bei Auftreten des Ereignisses *guthabenLeer1* der Eventhandler *guthabenKontrolle* aufzurufen ist

C#
```
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
```

Wichtig ist dabei die Verwendung des Operators += (siehe Sprachkapitel, Abschnitt 2.4), denn pro Ereignis sind durchaus mehrere Eventhandler möglich. In diesem Fall erfolgt deren Abarbeitung in der Reihenfolge der Anmeldung.

BEISPIEL 3.34: Zum Ereignis *guthabenLeer1* wird ein zweiter Eventhandler hinzugefügt

C#
Beim Eintreten des Ereignisses erscheint zunächst das vom ersten Eventhandler produzierte Meldungsfenster (siehe oben) und anschließend der Name des Kunden in einem Label.

```
// zweiten Eventhandler implementieren
private void kundenAnschrift(object o, string s)
{
    CKunde k = (CKunde) o;
    label3.Text = k.name;
}
 ....
// zweiten Eventhandler hinzufügen:
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(kundenAnschrift);
```

Falls ein Eventhandler nicht mehr benötigt wird, sollten Sie ihn wieder abmelden.

BEISPIEL 3.35: Abmelden eines Eventhandlers *kundenAnschrift*

C#
```
kunde1.guthabenLeer1 -= new CKunde.GuthabenLeer(kundenAnschrift);
```

Bemerkungen

Wenn Sie im Eigenschaften-Fenster der Visual Studio-Entwicklungsumgebung auf bekannte Weise Eventhandler für die Objekte der Bedienoberfläche erzeugen, so hat die IDE nicht nur den Rahmencode des Eventhandlers für Sie generiert, sondern – quasi im Verborgenen – auch die benutzten Ereignisse angemeldet. Üblicherweise übergeben diese Ereignisse zwei Parameter an die aufrufende Instanz: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse.

BEISPIEL 3.36: Rahmencode des automatisch generierten Eventhandlers für das *Click*-Ereignis eines *Button*

C#
```
private void button1_Click(object sender, EventArgs e)
{

}
```

Klappen Sie die Region *Vom Windows Form-Designer generierter Code* auf, so finden Sie die entsprechende Befehlszeile für die Anmeldung des Eventhandlers:

```
this.button1.Click += new EventHandler(this.button1_Click);
```

Die folgende Abbildung veranschaulicht nochmals die Syntax dieser Zeile.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Objekt Ereignis Delegat Event-Handler

Und zum Schluss noch ein Hinweis auf einen ziemlich häufigen Unterlassungsfehler:

HINWEIS: Wenn Sie die Codezeilen eines Eventhandlers komplett per Hand löschen (man sollte das eigentlich nicht tun), so müssen Sie auch die entsprechende Anmeldungszeile (siehe oben) löschen, ansonsten gibt es einen Compilerfehler!

3.5 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache wie C# realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

"Bis jetzt sind wir doch glänzend ohne Konstruktor ausgekommen!", werden Sie jetzt vielleicht einwenden. Ganz stimmt das nicht, denn wenn Sie sich um keinen eigenen Konstruktor kümmern, wird der von *System.Object* geerbte parameterlose *new*-Standardkonstruktor verwendet.

3.5.1 Konstruktor und Objektinitialisierer

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Name des Konstruktors ist immer identisch mit dem Namen der Klasse.

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts (*new*) aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *public void*-Methode hinzu, nur dass Sie der Methode den Namen der Klasse geben und auf das *void*-Schlüsselwort verzichten. Als Parameter übergeben Sie die Werte für die Felder, die initialisiert werden sollen.

SYNTAX:
```
public KlassenName(Datentyp Parameter)
{
   // Initialisierung der Klasse
}
```

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.37: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt

C#
```
public class CKunde
{
```

Die Felder:

```
    private string _anrede;
    private string _name;
    private int _plz;
    private string _ort;
    private bool _stammKunde;
    private decimal _guthaben;
```

Der erste Konstruktor initialisiert nur zwei Felder:

```
   public CKunde(string anr, string nam)
   {
       _anrede = anr; _name = nam;
   }
```

Der zweite Konstruktor initialisiert **alle** Felder der Klasse:

```
   public CKunde(string a, string n, int p, string o, bool s, decimal g)
   {
       _anrede = a;   _name = n;     _plz = p;
       _ort = o;  _stammKunde = s;   _guthaben = g;
   }
   ...
}
```

Aufruf

Nachdem Sie einer Klasse einen oder mehrere Konstruktoren hinzugefügt haben, sind Sie auch zur Verwendung von mindestens einem davon verpflichtet. Die bisher gewohnte einfache Instanziierung von Objekten ist nicht mehr möglich!

BEISPIEL 3.38: Zwei Objekte der Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert.

C#

Objekte referenzieren:

```
CKunde kunde1, kunde2, kunde3;
```

Sicherheitshalber bauen wir das Erzeugen der Objekte in einen Exception-Handler ein:

```
try
{
```

Für jedes Objekt wird ein anderer überladener Konstruktor verwendet:

```
    kunde1 = new CKunde("Herr", "Müller");
    kunde2 = new CKunde("Frau", "Hummel", 12345, "Berlin", true, 100);
    //  kunde3 = new CKunde();                        // erzeugt Compilerfehler!!!
    MessageBox.Show("Objekte erfolgreich erzeugt!");
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message + " Sch... Konstruktor!");
}
```

Wenn Sie den Code mit dem Anfangsbeispiel im Abschnitt 3.1.6 vergleichen, so sehen Sie, dass das Initialisieren der Objekte viel übersichtlicher geworden ist. Statt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt in einer einzigen Befehlszeile.

Objektinitialisierer

Objektinitialisierer erlauben das Erzeugen und Initialisieren von Instanzen auf Basis von (öffentlichen) Objekteigenschaften auch ohne das explizite Vorhandensein eines Konstruktors[1].

BEISPIEL 3.39: Erzeugen eines Objekts mittels Objektinitialisierer (das ist kein Konstruktoraufruf!)

C#

```
kunde1 = new CKunde {Anrede = "Herr", Name = "Müller};
```

Statischer Konstruktor

Neben einem oder mehreren "normalen" Konstruktoren kann eine Klasse auch über einen *statischen* Konstruktor verfügen. Ein solcher Konstruktor wird verwendet, um *static*-Felder (siehe 3.2.5) zu initialisieren oder um einmaligen Initialisierungscode auszuführen. Der Aufruf erfolgt automatisch, bevor die erste Instanz erstellt oder auf statische Klassenmitglieder verwiesen wird.

[1] Notwendig wurde diese Spracherweiterung vor allem wegen der neu eingeführten LINQ-Technologie, deren anonyme Typen beispielsweise nach einem solchen Feature verlangen (siehe Kapitel 6).

BEISPIEL 3.40: Eine Klasse besitzt die statische Eigenschaft *MWSt*, welche zu Beginn mit dem Wert 0,19 initialisiert werden soll.

C#
```
public class CAllerlei
{
    private static double _mwst;

    public static double MWSt
    {
        get { return (_mwst); }
        set { _mwst = value; }
    }

    static CAllerlei()                              // statischer Konstruktor
    {
        _mwst = 0.19;
    }
    ...
}
```

Der Zugriff von einem Formular *Form1* aus:

```
public partial class Form1 : Form
{
    ...
    textBox1.Text = CAllerlei.MWSt.ToString();       // zeigt 0,19
    ...
}
```

HINWEIS: Ein statischer Konstruktor akzeptiert weder Zugriffsmodifizierer, noch besitzt er Parameter, er kann auch nicht direkt aufgerufen werden.

3.5.2 Destruktor und Garbage Collector

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatzressourcen endet, ist der Destruktor für das Erledigen von "Aufräumarbeiten" zuständig, kurz bevor das Objekt sein Leben aushaucht.

In .NET haben wir allerdings keine echten Destruktoren, da hier die endgültige Zerstörung eines Objekts nicht per Code, sondern automatisch vom Garbage Collector vorgenommen wird. Dieser durchstöbert willkürlich und in unregelmäßigen Zeitabständen den Heap nach Objekten, um diejenigen zu suchen, die nicht mehr referenziert werden.

An die Stelle eines echten Destruktors tritt ein Quasi-Destruktor. Das ist eine Finalisierungsmethode, die zu einem unbestimmbaren Zeitpunkt vom Garbage Collector aufgerufen wird, kurz bevor dieser das Objekt vernichtet.

Ähnlich wie beim Konstruktor wird auch hier der Name der Klasse als Methodenbezeichner verwendet, allerdings mit einer Tilde (~) als Präfix. Der *public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:
```
~KlassenName()
{
    // hier Code für Aufräumarbeiten implementieren
}
```

BEISPIEL 3.41: Destruktor und Garbage Collector

C#

Unsere Klasse *CKunde* erhält ein öffentliches statisches Feld, welches durch den Konstruktor inkrementiert und durch den Quasi-Destruktor dekrementiert werden soll. Wir beabsichtigen damit, die Anzahl der momentan instanziierten Klassen (sprich Anzahl der Kunden) abzufragen.

Der auf das Wesentliche reduzierte Code von *CKunde*:

```
public class CKunde
{
     public static int anzahl = 0;

     // Konstruktor:
     public CKunde()
     {
         anzahl++;
     }

     // Destruktor:
      ~CKunde()
     {
          anzahl--;
     }
}
```

Wir verwenden zum Testen der Klasse ein Windows-Formular mit zwei *Button*s, einer *Timer*-Komponente (*Interval* = 1000, *Enabled* = *True*) und einem *Label*.

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
CKunde kunde1;  // Objekt referenzieren

// Objekt hinzufügen:
private void button1_Click(object sender, EventArgs e)
{
    kunde1 = new CKunde();
}

// Objekt entfernen:
private void button2_Click(object sender, EventArgs e)
{
```

BEISPIEL 3.41: Destruktor und Garbage Collector

```
        kunde1 = null;     // dereferenzieren
    }


    // Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:
    private void timer1_Tick(object sender, EventArgs e)
    {
        label1.Text = CKunde.anzahl.ToString();
    }
```

Ergebnis

Beim Programmtest müssen Sie etwas Geduld aufbringen.

Nach dem Programmstart fügen Sie durch Klicken auf den linken Button ein Objekt *kunde1* hinzu, wonach sich die Anzeige von 0 auf 1 ändert. Anschließend klicken Sie auf den rechten Button, um das Objekt wieder zu entfernen.

Es kann einige Zeit dauern, bis die Anzeige wieder auf 0 zurück geht, nämlich dann, wenn dem Garbage Collector gerade einmal wieder die Lust zum Aufräumen überkommt und er den Quasi-Destruktor aufruft[1].

Übrigens können Sie auch den linken Button mehrmals hintereinander klicken. Die Anzeige zählt zwar hoch, das aber täuscht, denn es bleibt bei nur einer Objektvariablen (*kunde1*). Allerdings wird Ressourcenverschwendung betrieben, denn dem Objekt wird immer wieder ein neuer Speicherbereich zugewiesen. Der vorher belegte Speicher liegt brach und wartet auf die Freigabe durch den Garbage Collector.

HINWEIS: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit dem Quasi-Destruktor ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode (*Close*- bzw. *Dispose*) oder *using* verwenden (siehe folgender Abschnitt).

3.5.3 Mit using den Lebenszyklus des Objekts kapseln

Mit dem Schlüsselwort *using* kann man nicht nur Namespaces einbinden (siehe Kapitel 5), sondern in völlig anderer Bedeutung auch für das sichere Erzeugen und Vernichten von Objekten sorgen. Hinter den Kulissen wird ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen. Das folgende (eigentlich lächerliche) Beispiel soll lediglich das Prinzip verdeutlichen.

BEISPIEL 3.42: Zum Prinzip von *using*

C#
```
using (CKunde kunde1 = new CKunde("Herr", "Müller"))      // Erzeugen des Objekts
{
    kunde1.Wohnort = "Berlin";                             // Arbeiten mit dem Objekt
    ...
}                                                          // Freigabe des Objekts
```

Weitaus sinnvollere Beispiele für *using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File (MMF) verwenden

und im Abschnitt 23.3.5 des ADO.NET-Kapitels (Datenbankzugriff).

3.5.4 Verzögerte Initialisierung

Erzeugen Sie wie bisher ein Objekt mit *new*, so wird der Speicher gleich bei der Initialisierung belegt. Die verzögerte Initialisierung (*Lazy Initialization*) von Objekten hat hingegen den Vorteil, dass die Objekte erst dann Speicherplatz belegen, wenn sie tatsächlich verwendet werden. Das lohnt sich besonders für Anwendungen mit sehr vielen oder sehr umfangreichen Klasseninstanzen.

Realisiert wird die verzögerte Initialisierung mit der ab .NET 4.0 eingeführten generischen Klasse *System.Lazy<>*, welche die tatsächliche Klasse kapselt.

Um das Prinzip zu verdeutlichen, gehen wir von einer allgemeinen Klasse aus:

```
public class Klasse1
{
    public Klasse1()
    {
        // Konstruktor
    }

    public string Eigenschaft1
    { get; set; }
```

```
    public void Methode1()
    {
        // ...
    }
}
```

Die verzögerte Initialisierung wird vorbereitet:

```
Lazy<Klasse1> objInit;
objInit = new Lazy<Klasse1>();
```

Bis jetzt wurde das Objekt noch nicht erstellt (der Wert der *objInit.IsValueCreated*-Eigenschaft ist *false*).

Erst beim Zugriff auf eines seiner Mitglieder wird das Objekt erzeugt:

```
objInit.Value.Eigenschaft1 = "Das ist der erste Wert!";
```

Damit ist das Objekt initialisiert (die *objInit.IsValueCreated*-Eigenschaft ist *true*).

HINWEIS: Standardwerte von Feldern werden erst bei der erstmaligen Verwendung zugewiesen, dasselbe gilt für die Ausführung eines evtl. vorhandenen Konstruktors.

3.6 Vererbung und Polymorphie

Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. C# unterstützt sowohl Vererbung als auch polymorphes Verhalten, da das Überschreiben (*Overriding*) der Basisklassenmethoden mit alternativen Implementierungen erlaubt ist.

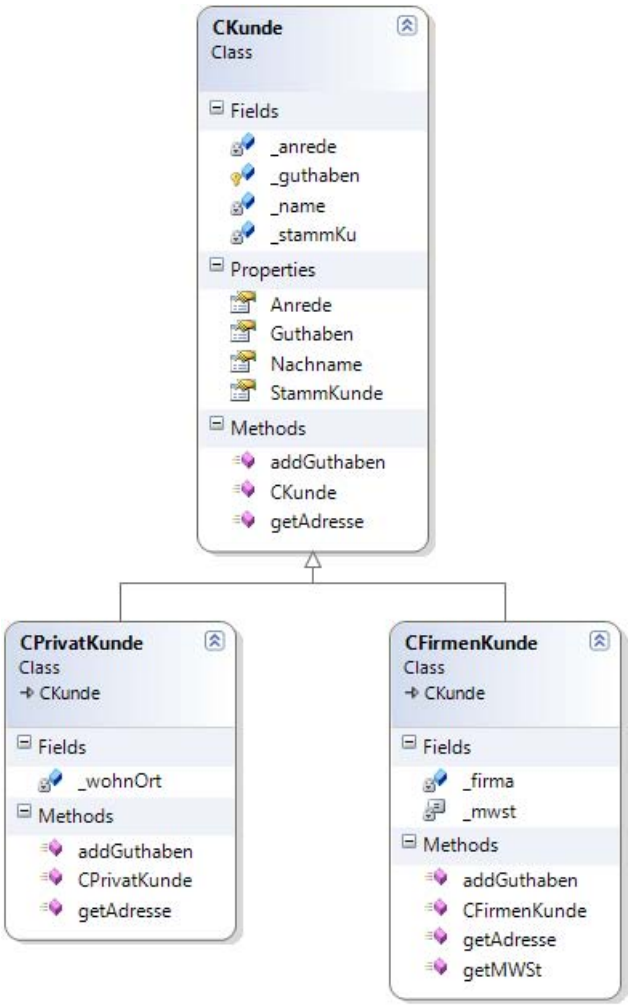
Durch Vererbung können Sie sich die Programmierarbeit wesentlich erleichtern, indem Sie spezialisierte Subklassen verwenden, die den Code zum großen Teil von einer allgemeinen Basisklasse erben. Die Subklassen heißen auch *abgeleitete Klassen, Kind-* oder *Unterklassen*, die Basisklasse wird auch als *Super-* oder *Elternklasse* bezeichnet. In den Subklassen können Sie bestimmte Funktionalitäten überschreiben, um spezielle Prozesse auszuführen.

Lassen Sie uns anhand eines kurzen und dennoch ausführlichen Beispiels die wichtigsten Vererbungstechniken demonstrieren! Wir beginnen mit dem Klassendiagramm.

3.6.1 Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.

Das folgende, mit Visual Studio erzeugte, Klassendiagramm[1] zeigt eine Basisklasse *CKunde*, von der die Klassen *CPrivatKunde* und *CFirmenKunde* "erben". Die Basisklasse hat die Eigenschaften *Anrede*, *Nachname*, *StammKunde* (ja/nein) und *Guthaben* und die Methoden *getAdresse()* und *addGuthaben()* (das Guthaben ist hier als Bonus zu verstehen, der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird). Die Methode *CKunde* ist nichts weiter als der Konstruktor.



[1] Der Umgang mit dem Klassendesigner wird ausführlich im Kapitel 26 beschrieben.

3.6.2 Method-Overriding

Die Subklassen *CPrivatKunde* und *CFirmenKunde* können auf sämtliche Eigenschaften und Methoden der Basisklasse zugreifen und fügen selbst eigene Methoden (auch Eigenschaften wären natürlich möglich) hinzu.

Die "geerbten" Methoden *getAdresse* und *addGuthaben* sind in unserem Beispiel allerdings so genannte *überschriebene Methoden* (*Method-Overriding*), d h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

3.6.3 Klassen implementieren

Vorbild für die drei zu implementierenden Klassen ist obiges Klassendiagramm.

Basisklasse CKunde

Die Deklaration entspricht (fast) der einer normalen Klasse. Dass es sich um eine Basisklasse handelt, erkennt man in unserem konkreten Fall eigentlich nur an dem *protected*-Feld und an den *virtual*-Methodendeklarationen[1].

```
public class CKunde                // Basisklasse
{
```

Die privaten Felder:

```
    private string _anrede;
    private string _name;
    private bool _stammKu;
```

Durch den *protected*-Modifizierer für das *guthaben*-Feld wird es möglich, dass auch die beiden Subklassen auf dieses Feld zugreifen können:

```
    protected decimal _guthaben = 0;      // Feld ist in Subklassen sichtbar!
```

Ein eigener Konstruktor ersetzt den Standardkonstruktor:

```
    public CKunde(string anr, string nName)     // Konstruktor
    {
        _anrede = anrede; _name = nName;
    }
```

Die Eigenschaften:

```
    public string Anrede
    {
         get { return (_anrede); }
         set { _anrede = value; }
    }
```

[1] Eigentlich hätten wir unsere Klasse auch noch als *abstract* deklarieren müssen (siehe dazu Abschnitt 3.7 1).

```
    public string Nachname
    {
         get { return (_name); }
         set { _name = value; }
    }
    public bool StammKunde
    {
        get {return(_stammKu); }
        set {_stammKu = value; }
    }

    public decimal Guthaben        // ReadOnly
    {
        get {return(_guthaben); }
    }
```

Nun zu den beiden Methoden, die durch die Subklassen überschrieben werden können.

Der Rückgabewert der virtuellen Methode *getAdresse()* setzt sich aus Anrede und Namen des Kunden zusammen:

```
    public virtual string getAdresse()                    // virtuelle Methode
    {
        string s = _anrede + " " + _name;
        return(s);
    }
```

Die virtuelle Methode *addGuthaben()* erhöht das Guthaben des Kunden um den im Argument übergebenen Bonusbetrag. Allerdings kommen nur Stammkunden in diesen Genuss:

```
    public virtual void addGuthaben(decimal betrag)      // virtuelle Methode
    {
        if (_stammKu) _guthaben += betrag;
    }
}
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *override*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *virtual* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

```
public class CPrivatKunde : CKunde             // erbt von der Basisklasse CKunde!
{
   private string _wohnOrt;
```

Der Konstruktor ist unbedingt notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet. Es wird das *base*-Schlüsselwort benutzt, um den Konstruktor der Basisklasse aufzurufen.

```
    public CPrivatKunde(string anrede, string name, string ort): base(anrede, name)
    {
        _wohnOrt = ort;              // klassenspezifische Ergänzung
    }
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird.

```
    public override string getAdresse()
    {
        const char LF = (char) 10;       // Zeilenvorschub
        return(base.adresse() + LF + _wohnOrt);
    }
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    public override void addGuthaben(decimal geld)
    {
        // Zugriff auf protected-Variable in Basisklasse:
        _guthaben += 0.05M * geld;
    }
}
```

Subklasse CFirmenKunde

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in folgenden Details von der Klasse *CPrivatKunde*:

- Die Methode *getAdresse()* liefert statt des Wohnorts den Namen der Firma des Kunden.
- Die *addGuthaben()*-Methode berechnet zunächst den Nettobetrag und addiert davon 1% zum Bonusguthaben. Damit nur Stammkunden in den Genuss dieser Vergünstigung kommen, wird dazu die gleichnamige Methode der Basisklasse aufgerufen.
- Die neu hinzugekommene "stinknormale" Methode *getMWSt()* erlaubt einen Lesezugriff auf die Mehrwertsteuer-Konstante.

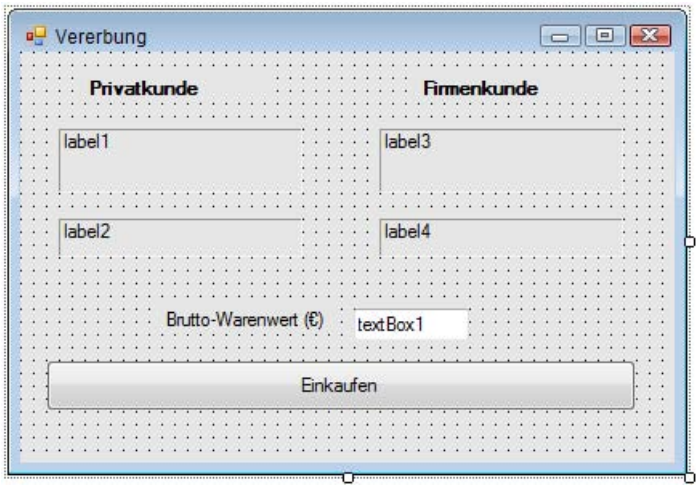
```
public class CFirmenKunde : CKunde
{
    private string _firma;
    private  const float _mwst = 0.19F;     // Mehrwertsteuer

    public CFirmenKunde(string anrede, string name, string frm): base(anrede, name)
    {
        _firma = frm;
    }

    public override string adresse()
    {
        const char LF = (char) 10;      // Zeilenvorschub
```

```
        return(base.adresse() + LF + _firma);
    }

    public override void addGuthaben(decimal brutto)
    {
        decimal netto = brutto / Convert.ToDecimal(1 + _mwst);
        base.addGuthaben(netto * 0.01M);   // Aufruf der Methode der Basisklasse
    }

    public double getMWSt()    // eine ganz normale Methode
    {
        return(_mwst);
    }
}
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie die folgende Benutzerschnittstelle:



3.6.4 Implementieren der Objekte

Für einen kleinen Test genügt es, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten.

```
CPrivatKunde kunde1;
CFirmenKunde kunde2;
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

```
public partial class Form1 : Form
{ ...

    public Form1()
    {
        InitializeComponent();

        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.StammKunde = false;

        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
        textBox1.Text = "100";
    }
```

Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

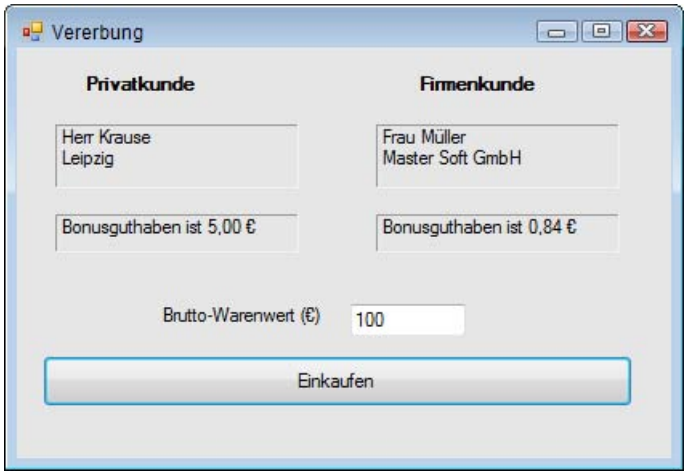
```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = kunde1.getAdresse();
        kunde1.addGuthaben(brutto);
        label2.Text = "Bonusguthaben ist " + kunde1.Guthaben.ToString("C");

        label3.Text = kunde2.getAdresse();
        kunde2.addGuthaben(brutto);
        label4.Text = "Bonusguthaben ist " + kunde2.guthaben.ToString("C");
    }
}
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.

Die Werte in der folgenden Laufzeitabbildung sind wie folgt zu interpretieren:

- Dem Privatkunden Krause wurde ein Guthaben von 5 € (5% aus 100 €) zugebilligt (Stammkundschaft spielt bei Privatkunden keine Rolle, da die Methode *addGuthaben()* komplett überschrieben ist).
- Frau Müller ist eine Firmenkundin und erhält – nur weil sie Stammkundin ist – ein mickriges Guthaben von 0,84 € (1% auf den Nettowert).
- Durch wiederholtes Klicken auf "Einkaufen" kumulieren die Bonusguthaben.



3.6.5 Ausblenden von Mitgliedern durch Vererbung

Durch in eine abgeleitete Klasse oder Struktur eingeführte Mitglieder (Konstanten, Felder, Eigenschaften, Methoden, Ereignisse oder Typen) werden alle gleichnamigen Basisklassenelemente verdeckt bzw. ausgeblendet.

BEISPIEL 3.43: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

C#

```
class CKunde                        // Basisklasse
{
   ...
   public void test()
   {
        MessageBox.Show("Hallo Kunde!");
   }
}
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatKunde* hinzu:

```
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *"WindowsFormsApplication1.CPrivatKunde.test()" blendet den versteckten Member "WindowsFormsApplication1.CKunde.test()" aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.*

BEISPIEL 3.43: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

Der Test erfolgt in *Form1*:

```
public partial class Form1 : Form
{
    private CPrivatKunde kunde1 = new CPrivatKunde();
    ...
    private void button1_Click(object sender, EventArgs e)
    {
       ...
       kunde1.test();
    }
 }
```

Ergebnis

Wie Sie sehen, wurde die Methode *test()* der Klasse *CKunde* durch die Methode *test()* der Klasse *CPrivatKunde* ausgeblendet:

Um Missverständnissen vorzubeugen (und um obigen Warnhinweis im Quellcode zu vermeiden), sollte man die gleichnamige Methode in der abgeleiteten Klasse mit dem *new*-Schlüsselwort markieren.

BEISPIEL 3.44: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

C#

```
...
class CPrivatKunde : CKunde     // abgeleitete Klasse
{
   ...
   new public void test()
   {
     MessageBox.Show("Hallo PrivatKunde!");
   }
}
```

HINWEIS: Die Deklaration eines neuen Mitglieds verdeckt ein geerbtes Mitglied nur innerhalb des Gültigkeitsbereichs des neuen Mitglieds!

3.6.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennengelernt haben, werden wir auch die folgenden Regeln und Hinweise verstehen:

- Alle öffentlichen Eigenschaften und Methoden der Basisklasse sind auch über die abgeleiteten Subklassen verfügbar.
- Methoden der Basisklasse, die von den abgeleiteten Subklassen überschrieben werden dürfen (so genannte *virtuelle Methoden*), müssen mit dem Schlüsselwort *virtual* deklariert werden.
- Fehlt das Schlüsselwort *virtual* bei der Methodendeklaration, so bedeutet das, dass dies die einzige Implementierung der Methode ist.
- Methoden der Subklassen, welche die gleichnamige Methode der Basisklasse überschreiben, müssen mit dem Schlüsselwort *override* deklariert werden.
- Wenn Sie das *override*-Schlüsselwort in der Subklasse vergessen, wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.
- Private Felder der Basisklasse, auf die die Subklassen zugreifen dürfen, müssen mit *protected* deklariert werden.
- Die Basisklasse wird der Subklasse durch einen der Klassendeklaration nachgestellten Doppelpunkt bekannt gemacht:

  SYNTAX:
  ```
  class SubKlasse : BasisKlasse
  {
      // ... Implementierungscode
  }
  ```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *base*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *this*-Objekt auf die eigene Klasse.
- Wenn die Basisklasse einen eigenen Konstruktor verwendet, so müssen in den Subklassen ebenfalls eigene Konstruktoren definiert werden (Konstruktoren können nicht vererbt werden!).
- Der Konstruktor einer Subklasse muss den Konstruktor seiner Basisklasse aufrufen (*base*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Wenn Sie mit Vererbung arbeiten, sollten Sie Folgendes beachten:

- Es gibt keinerlei Beschränkung bezüglich der Stufenanzahl der Vererbungshierarchie. Sie können die Hierarchie so tief wie nötig staffeln, die Eigenschaften/Methoden werden trotzdem durch alle Vererbungsstufen hindurchgereicht. Allgemein gilt, je weiter unten sich eine Klasse in der Hierarchie befindet, umso spezialisierter ist ihr Verhalten. Zum Beispiel eine *CHoch-*

schulKunden-Klasse, die von einer *CSchulKunden* erbt und diese wiederum von der *CKunden*-Klasse.

- Um die Komplexität zu minimieren und die Wartbarkeit des Codes zu vereinfachen, sollten Sie die Vererbungshierarchie nicht tiefer als ca. vier Stufen staffeln.
- Jede Subklasse kann nur von einer Basisklasse erben! So kann z.B. eine *CHochSchulKunden*-Klasse nicht sowohl von der *CKunden*-Klasse und einer *CSchulKunden*-Klasse erben. Das ist in Ordnung so, denn eine solche multiple Vererbung könnte sehr schnell zu einem komplexen, unübersichtlichen und nicht mehr beherrschbaren Ungetüm entarten.

Es gibt zwei primäre Anwendungsfälle für Vererbung in Anwendungen:

- Sie verwenden Objekte unterschiedlichen Typs mit ähnlicher Funktionalität. So erben z.B. *CSchulKunden*-Klasse und *CStaatsKunden*-Klasse von der *CKunden*-Klasse.
- Sie haben gleiche Prozesse mit einer Menge von Objekten auszuführen. So erbt z.B. jeder Typ eines Geschäftsobjekts von einer Business Object(BO)-Klasse.

Sie sollten in folgenden Fällen auf Vererbung verzichten:

- Sie brauchen nur eine einzige Funktion von der Basisklasse. In diesem Fall sollten Sie die Funktion in die eigene Klasse delegieren, statt von einer anderen zu erben.
- Sie möchten alle Funktionen überschreiben. In einem solchen Fall sollten Sie eine Schnittstelle (*Interface*, siehe Abschnitt) statt Vererbung verwenden.

3.6.7 Polymorphes Verhalten

Untrennbar mit der Vererbung verbunden ist die so genannte Polymorphie (Vielgestaltigkeit). Polymorphes Verhalten bedeutet, dass erst zur Laufzeit einer Anwendung entschieden wird, welche der möglichen Methodenimplementierungen aufgerufen wird, da dies zum Zeitpunkt des Compilierens noch unbekannt ist.

Im obigen Beispiel hatten wir von den Vorzügen der Polymorphie allerdings noch keinen Gebrauch gemacht, denn Privat- und Firmenkunde wurden in einzelnen Objektvariablen gespeichert und bereits per Programmcode fest mit ihren Methoden *getAdresse()* und *addGuthaben()* verbunden.

Um Polymorphie sichtbar zu machen, müssen wir das bei der Implementierung der Objekte zielgerichtet ausnutzen. Wie wir gleich sehen werden, treten die Vorzüge von Polymorphie besonders augenscheinlich zutage, wenn Objekte unterschiedlicher Klassenzugehörigkeit nacheinander in Arrays oder Auflistungen abgespeichert werden.

BEISPIEL 3.45: Polymorphes Verhalten

C#

Wir nehmen die drei Klassen des Vorgängerbeispiels (*CKunde*, *CPrivatKunde*, *CFirmenKunde*) als Grundlage. An deren Implementierungen brauchen wir keinerlei Veränderungen vorzunehmen, denn polymorphes Verhalten ergibt sich als logische Konsequenz aus der Vererbung von Klassen. Änderungen müssen wir lediglich beim Abspeichern der Objektvariablen vornehmen.

BEISPIEL 3.45: Polymorphes Verhalten

Aus den Subklassen *CPrivatKunde* und *CFirmenKunde* wollen wir insgesamt drei Objekte (*kunde1*, *kunde2*, *kunde3*) instanziieren (ein Privatkunde, zwei Firmenkunden).

Innerhalb des Klassencodes von *Form1* deklarieren Sie:

```
private CPrivatKunde kunde1;
private CFirmenKunde kunde2, kunde3;
private CKunde[] kunden = new CKunde[3];       // Array für 3 Objekte!

private const char LF = (char) 10;             // für Zeilenvorschub
```

Im Konstruktor von *Form1* werden die notwendigen Initialisierungen vorgenommen:

```
    public Form1()
    {
        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.StammKunde = false;
        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.StammKunde = true;
        kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");
        kunde3.StammKunde = false;
```

Da das Array vom Typ der Basisklasse ist, kann es auch Objekte der Subklassen (Privat- und Firmenkunden) in wahlloser Reihenfolge aufnehmen:

```
        kunden[0] = kunde1;
        kunden[1] = kunde2;
        kunden[2] = kunde3;
        textBox1.Text = "100";
}
```

Das Array wird in einer *for*-Schleife durchlaufen und ausgelesen. Dabei werden die polymorphen Methoden (das sind die mit *virtual* bzw. *override* deklarierten) für alle Objekte aufgerufen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = String.Empty;
        for (int i = 0; i < kunden.Length; i++)
        {
            kunden[i].addGuthaben(brutto);
            label1.Text = label1.Text + LF + kunden[i].getAdresse() + LF
                          + kunden[i].Guthaben.ToString("C");
        }
    }
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sein können, "weiß" das Programm zur Laufzeit genau, welche Implementierung der beiden polymorphen Methoden *getAdresse()* und *addGuthaben()* jeweils für Privat- und für Firmenkunden die richtige ist.

HINWEIS: Genau hier liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.46: Polymorphes Verhalten Variante 2

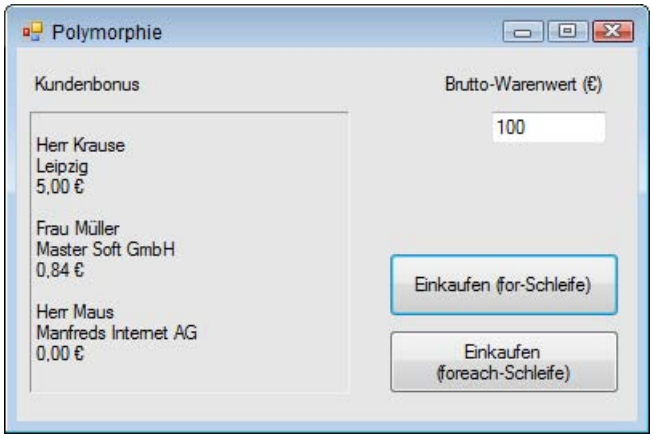
C#

Eine alternative Implementierung mittels *foreach*-Schleife bringt die Polymorphie noch deutlicher ans Tageslicht, da die Methodenaufrufe nicht mit Objekten der Subklassen *CPrivatKunde*/*CFirmenkunde*, sondern mit Objekten der Basisklasse *CKunde* verknüpft sind:

```
private void button2_Click(object sender, EventArgs e)
{
    decimal brutto = Convert.ToDecimal(textBox1.Text);
    label1.Text = String.Empty;
    foreach (CKunde ku in kunden)
    {
        ku.addGuthaben(brutto);
        label1.Text = label1.Text + LF + ku.getAdresse() + LF +
                                              ku.Guthaben.ToString("C");
    }
}
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verbunden sind. Egal ob Privat- oder Firmenkunde – es werden immer die jeweils passenden Methodenimplementierungen aufgerufen[1].



Das tiefere Verständnis der Polymorphie ist mit Sicherheit der schwierigste Part der OOP, deshalb wurde unser Beispiel bewusst einfach gehalten, damit Sie zunächst zu einem Grundverständnis gelangen, welches Sie später weiter ausbauen können.

[1] Tja, der arme Herr Maus. Weil er kein Stammkunde ist, bekommt er auch kein Bonusguthaben.

3.6.8 Die Rolle von System.Object

Jedes Objekt in .NET erbt von der Basisklasse *System.Object*. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

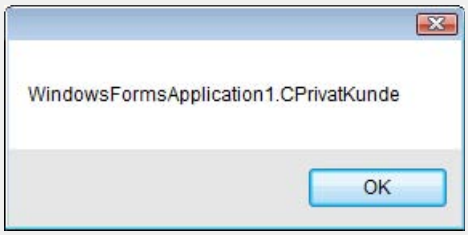
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird es mit diesem Konstruktor erzeugt.

Viele der öffentlichen Eigenschaften und Methoden von *System.Object* haben eine Standardimplementation. Das heißt, Sie brauchen selbst keinerlei Code zu schreiben, um sie zu verwenden.

BEISPIEL 3.47: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier *Vererbung1*) und die Klassenzugehörigkeit von *kunde1*.

C#
```
MessageBox.Show(kunde1.ToString());
```

Ergebnis



Sie können das standardmäßige Verhalten von *ToString()* mittels *override*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.48: Die gleiche *ToString()*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

C#
```
public override string ToString()
{
   return (_name);
}
```

Ergebnis

3.7 Spezielle Klassen

Es gibt einige Klassen, die spezielle Features aufweisen, bzw. nur für einen beschränkten Einsatzbereich infrage kommen. Gekennzeichnet werden diese Klassen in der Regel durch einen vorangestellten Modifizierer (*abstract*, *sealed*, *partial*, *static*).

3.7.1 Abstrakte Klassen

Klassen, die lediglich ihr "Erbmaterial" an andere Klassen weitergeben und von denen selbst keine Instanzen gebildet werden, bezeichnet man als *abstrakt*. Typische Beispiele für abstrakte Klassen wären *Fahrzeug*, *Tier* oder *Nahrung*[1].

Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, müssen diese mit dem Modifikator *abstract* gekennzeichnet werden.

BEISPIEL 3.49: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

C#
```
public abstract class CKunde
{ ... }
```

Während die Referenzierung nach wie vor möglich ist

```
CKunde kunde;
```

schlägt der Versuch einer Instanziierung fehl:

```
kunde = new CKunde("Herr", "Krause");              // Fehler
```

Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, den Schnittstellen (Interfaces). Eine Schnittstelle müssen Sie sich wie eine abstrakte Klasse vorstellen, in welcher nur öffentliche Methoden definiert, aber nicht implementiert werden (die Methodenrümpfe bleiben leer). Die Deklaration einer Schnittstelle ähnelt der einer Klasse, nur dass das Schlüsselwort *class* gegen das Schlüsselwort *interface* ausgetauscht wird (siehe Abschnitt 3.8).

Abstrakte Methoden

In Verbindung mit polymorphem Verhalten finden sich innerhalb abstrakter Klassen oft auch *abstrakte Methoden*, diese enthalten grundsätzlich keinen Code, da sie in den abgeleiteten Klassen komplett mit *override* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *abstract*. Die Deklaration erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.50: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett implementiert.

C#
```
public abstract class CKunde
{
    ...
```

[1] Können vielleicht Sie sich vorstellen, wie eine Instanz von *Nahrung* aussehen soll?

BEISPIEL 3.50: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett implementiert.

```
    public abstract string adresse();            // abstrakte Methode
    ...
}
```

Da eine abstrakte Methode implizit eine virtuelle Methode darstellt, muss sie in der Subklasse mit *override* deklariert werden.

```
public class CPrivatKunde : CKunde;
{
    ...
    public override string adresse()          // überschreibt abstrakte Methode
    {
         return(_anrede + "  " + _name + "  " & _wohnort)
    }
    ...
}
```

3.7.2 Versiegelte Klassen

Wenn Sie unbedingt verhindern möchten, dass andere Programmierer von einer von Ihnen entwickelten Komponente weitere Subklassen ableiten, so müssen Sie Ihre Klasse mit Hilfe des Modifikators *sealed* schützen.

BEISPIEL 3.51: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

C#
```
public sealed class CPrivatKunde : CKunde
{    ...
}
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
public class CStudent : CPrivatKunde          // Fehler!!!
{
...
}
```

HINWEIS: Vererbungsmodifikatoren wie *abstract* und *virtual* führen in einer versiegelten Klasse zum Compilerfehler, da sie keinen Sinn ergeben!

Übrigens: Ein bekanntes Beispiel für eine versiegelte Klasse ist der *string*-Datentyp, was jedweden Begehrlichkeiten einen Riegel vorschiebt.

3.7.3 Partielle Klassen

Das Konzept partieller Klassen ermöglicht es, den Quellcode einer Klasse auf mehrere einzelne Dateien aufzusplitten. In Visual Studio wird zum Beispiel auf diese Weise der vom Designer automatisch erzeugte Layout-Code vom Code des Entwicklers getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms Projekts selbst überzeugen kann (siehe auch neues Code-Beside-Modell von ASP.NET).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *partial* zu kennzeichnen, dieser muss hinter dem Sichtbarkeitsmodifizierer (*private, public, protected*, ...) stehen.

BEISPIEL 3.52: Eine einfache Klasse *CKunde*

C#

```
public class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }

        public string Name
        { get { return (_name); } }

        public decimal Guthaben
        { get { return (_guthaben); } }

        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
...
```

Die Klasse *CKunde* könnte (als eine von vielen Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
public partial class CKunde
{
        private string _name;
        protected decimal _guthaben = 0;

        public CKunde(string nachName)
        { _name = nachName; }
}

public partial class CKunde
{
        public string Name
        { get { return (_name); } }
```

BEISPIEL 3.52: Eine einfache Klasse *CKunde*

```
        public decimal Guthaben
        { get { return (_guthaben); } }
}

public partial class CKunde
{
        public void addGuthaben(decimal betrag)
        { _guthaben += betrag; }
}
```

3.7.4 Statische Klassen

Mit dem *static*-Modifizierer kann man nicht nur statische Klassenmitglieder (Eigenschaften, Methoden, siehe 3.2.5 und 3.3.3), sondern auch komplette *statische* Klassen deklarieren. Eine solche Klasse kann nicht instanziiert werden und hat nur statische Mitglieder. Praktische Anwendungen ergeben sich z.B. für Formelsammlungen. Auch in der .NET-Klassenbibliothek finden sich zahlreiche vordefinierte statische Klassen (z.B. *File*, *Directory*, *Debug*, ...).

BEISPIEL 3.53: Eine statische Klasse zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt)

C#

```
public static class CKugel
{
    private static double Pi = Math.PI;

    public static double Durchmesser_Volumen(double durchmesser)
    {
        double vol = Math.Pow(durchmesser, 3) * Pi/6.0;
        return vol;
    }

    public static double Volumen_Durchmesser(double volumen)
    {
        double dur = Math.Pow(6/Pi * volumen, 1/3.0);
        return dur;
    }
}
```

Beispielhafte Anwendung der statischen Klasse *CKugel* zur Berechnung des Durchmessers einer Kugel mit 1 Kubikmeter Rauminhalt:

```
double v = 1.0d;
double d = CKugel.Volumen_Durchmesser(v);          // liefert 1,24...
```

Die Verwendung einer ähnlichen Klasse wird im folgenden Praxisbeispiel demonstriert:

▶ 3.9.2 Eine statische Klasse anwenden

3.8 Schnittstellen (Interfaces)

Das .NET-Framework (die CLR) unterstützt keine Mehrfachvererbung, d h., eine Unterklasse kann immer nur von einer einzigen Oberklasse erben. Dies ist wohl mehr ein Segen als ein Fluch, denn allzu leicht würde sonst der Programmierer im Gestrüpp mehrfacher Vererbungsbeziehungen über mehrere Hierarchie-Ebenen hinweg die Übersicht verlieren, instabiler Code und Chaos wären die Folge.

Ein Ausweg ist die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile. Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.7.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

3.8.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Statt des Schlüsselworts *class* verwenden Sie aber *interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle mit "I" beginnen.

BEISPIEL 3.54: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert

C#
```
public interface IPerson
{
    string Nachname
    { get; set; }

    string Vorname
    { get; set; }

    string getName();
}
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (*public string Nachname ...*), diese haben aber in einer Schnittstellendefinition generell nichts zu suchen.

HINWEIS: Die Festlegung der Zugriffsmodifizierer für die Mitglieder der Schnittstelle ist allein Angelegenheit der Klasse, die die Schnittstelle implementiert!

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

3.8.2 Implementieren einer Schnittstelle

Die Syntax entspricht der bei der normalen Implementierurngsvererbung[1], d h., an die Deklaration der erbenden Klasse wird ein Doppelpunkt angefügt, dem der Namen der Schnittstelle folgt.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.55: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

C#

```
class CKunde : IPerson
{
       private string _nachname;
       private string _vorname;
       ...
```

Alle von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
       public string Nachname
       {
           get { return _nachname; }
           set { _nachname = value; }
       }

       public string Vorname
       {
           get { return _vorname; }
           set { _vorname = value; }
       }

       public override string  getName()
       {
          return _vorname + " " +_nachname ;
       }
```

Es folgen die normalen Klassenmitglieder:

```
       ...
}
```

Den kompletten Code finden Sie Im Praxisbeispiel

► 3.9.4 Schnittstellenvererbung verstehen

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

[1] Der Vergleich trifft zumindest dann zu, wenn nur eine einzige Schnittstelle implementiert wird.

3.8.3 Abfragen, ob Schnittstelle vorhanden ist

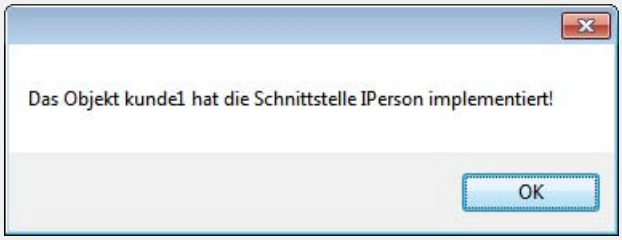
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *is*-Operator.

BEISPIEL 3.56: Abfragen, ob das Objekt *kunde1* die Schnittstelle *IPerson* implementiert hat

C#

```
private void button2_Click(object sender, EventArgs e)
{
  if (kunde1 is IPerson)
      MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!");
}
```

Ergebnis



3.8.4 Mehrere Schnittstellen implementieren

Eine Klasse kann nicht nur eine, sondern auch mehrere Schnittstellen gleichzeitig implementieren, was quasi Mehrfachvererbung bedeutet, wie sie mit der klassischen Implementierungsvererbung unmöglich ist.

BEISPIEL 3.57: Eine Klasse implementiert zwei Schnittstellen

C#

```
public class CPrivatkunde : IPerson, IKunde
{
    ...
}
```

3.8.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Statt von einer abstrakten Klasse zu erben, werden die abstrakten Methoden über eine Schnittstelle veröffentlicht. Damit erlangt man gewissermaßen die Funktionalität der Mehrfachvererbung und umgeht deren Nachteile.
- Eine Schnittstelle ist wie ein Vertrag: Sobald eine Klasse eine Schnittstelle implementiert, muss sie auch ausnahmslos alle (!) Mitglieder der Schnittstelle implementieren und veröffentlichen.

- Der Name der implementierten Methode sowie deren Signatur muss mit deren Definition in der Schnittstelle exakt übereinstimmen.
- Mehrere Schnittstellen können zu einer neuen Schnittstelle zusammengefasst werden und selbst wieder Schnittstellen implementieren.

HINWEIS: Mehr zur Schnittstellenprogrammierung finden Sie beispielsweise im Kapitel 5 (*IEnumerable*-Interface) oder im Kapitel 25 (Klassen-Designer).

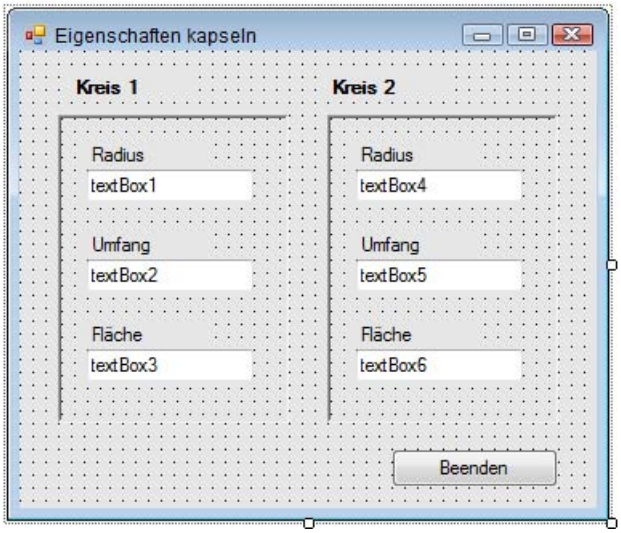
3.9 Praxisbeispiele

3.9.1 Eigenschaften sinnvoll kapseln

Das Deklarieren von Eigenschaften als öffentliche Variablen der Klasse heißt immer, das Brett an der dünnsten Stelle zu bohren. Der fortgeschrittene Programmierer verwendet stattdessen sogenannte Property-Methoden, die einen kontrollierten Zugriff erlauben. Außerdem ermöglichen die Property-Methoden auch die Implementierung von *berechneten Eigenschaften*, die aus den (privaten) Zustandsvariablen ermittelt werden. Im vorliegenden Rezept handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *radius*, *umfang* und *fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *r*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen großen Vorteil der OOP zu demonstrieren (ohne viel Mehraufwand lassen sich beliebig viele Instanzen einer Klasse erzeugen), wollen wir mit zwei Objekten (*kreis1* und *kreis2*) arbeiten.



Quellcode (CKreis)

Unsere Klasse wird außerhalb von *Form1* definiert, wir werden sie sogar in ein separates Klassenmodul auslagern. Wählen Sie das Menü *Projekt|Klasse hinzufügen...* und geben Sie dem Klassenmodul den Namen *CKreis.cs*.

Deklarieren und implementieren Sie nun die Klasse wie folgt:

```
public class CKreis
{
  private double r;                    // das einzige Feld (Zustandsvariable)
```

Die Rückgabewerte der Eigenschaften sind hier vom *string*-Datentyp, um die Bedienoberfläche von den lästigen Konvertierungen *string* in *double* und umgekehrt zu entlasten.

Die Eigenschaft *radius*:

```
  public string radius
  {
     get {return (r.ToString("#,#0.00")); }
     set
     {
       if (value != "")  r = Convert.ToDouble(value);
       else r = 0;
     }
  }
```

Die Eigenschaft *umfang*:

```
  public string umfang
  {
     get {return (2 * Math.PI * r).ToString("#,#0.00"); }
     set
     {
       if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
       else r = 0;
     }
  }
```

Die Eigenschaft *fläche*:

```
  public string fläche
  {
     get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
     set
     {
       if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
       else r = 0;
     }
  }
```

Der eigene Konstruktor:

```
  public CKreis(double rad)
```

```
   { r = rad;  }
}
```

Quellcode (Form1)

Wechseln Sie in den Klassencode von *Form1*.

```
public partial class Form1 : Form
{  ...
```

Ein Objekt *kreis1* wird erzeugt und kann (ein Dankeschön an den Konstruktor) gleich mit dem Radius 1.0 (Maßeinheit soll hier keine Rolle spielen) initialisiert werden:

```
CKreis kreis1 = new CKreis(1.0);
```

Die folgenden Eventhandler sind einfach und übersichtlich, da die Objekte ihre inneren Funktionalitäten kapseln:

```
    private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.radius = textBox1.Text;
        textBox2.Text = kreis1.umfang;
        textBox3.Text = kreis1.fläche;
    }

    private void textBox2_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.umfang = textBox2.Text;
        textBox1.Text = kreis1.radius;
        textBox3.Text = kreis1.fläche;
    }

    private void textBox3_KeyUp(object sender, KeyEventArgs e)
    {
        kreis1.fläche = textBox3.Text;
        textBox1.Text = kreis1.radius;
        textBox2.Text = kreis1.umfang;
    }
    ...
}
```

Der Code für das zweite Objekt (*kreis2*) ist völlig analog zu *kreis1*, sodass hier auf die Wiedergabe des Quellcodes verzichtet werden kann (siehe Beispieldaten zum Buch).

Test

Sobald Sie eine beliebige Eigenschaft ändern, werden die anderen zwei sofort aktualisiert! Wegen der in der Klasse eingebauten Eingabeprüfung verursacht ein leerer Eingabewert keinen Fehler.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Bemerkungen

- Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.
- Ab C# 3.0 kann man ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen passenden Konstruktor gibt. Sie könnten also auf den Konstruktor im Code von *CKreis* verzichten und stattdessen im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften (nicht private Zustandsvariablen!) mittels eines Objektinitialisierers wie folgt vornehmen:

```
CKreis kreis1 = new CKreis { radius = "1.0" };
```

3.9.2 Eine statische Klasse anwenden

Statische Klassen eignen sich ideal für Formelsammlungen (siehe z.B. *Math*-Klasse), da keine Objekte erzeugt werden müssen, denn es kann gleich "losgerechnet" werden. Das vorliegende Rezept demonstriert dies anhand einer statischen Klasse *CKugel* zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt)[1].

$V = 4/3 * Pi * r^3$

Nimmt man anstatt des Radius den Durchmesser *d* der Kugel, so ergibt sich daraus nach einigen Umstellungen die folgende Berechnungsformel für das Volumen *V*:

$V = d^3 * Pi/6$

Oberfläche

Lediglich ein Formular mit zwei Textfeldern zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitansicht).

Quellcode

```
public static class CKugel
// public class CKugel          // das würde auch gehen!
{
        public static double Durchmesser_Volumen(string durchmesser)
        {
            double dur = System.Double.Parse(durchmesser);
            double vol = Math.Pow(dur, 3) * Math.PI/6.0;
            return vol;
        }

        public static double Volumen_Durchmesser(string volumen)
        {
            double vol = System.Double.Parse(volumen);
            double dur = Math.Pow(6/Math.PI * vol, 1/3.0);
```

[1] Alter Merkspruch zum Kugelvolumen: "Bedächtig kommt sie einher geschritten – vier Drittel Pi mal R zur Dritten!"

```
            return dur;
        }
}
```

Die Verwendung der Klasse im Formularcode:

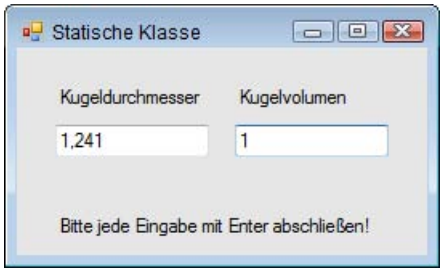
```
public partial class Form1 : Form
{  ...
```

Die Berechnung startet nach Betätigen der Enter-Taste:

```
        private void textBox1_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox1.Text != ""))
            {
                textBox2.Text = CKugel.Durchmesser_Volumen(textBox1.Text).ToString("#,##0.000");
            }
        }

        private void textBox2_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox2.Text != ""))
            {
                textBox1.Text = CKugel.Volumen_Durchmesser(textBox2.Text).ToString("#,##0.000");
            }
        }
}
```

Test

Es ist egal, ob Sie den Radius oder das Volumen eingeben. Nach Betätigen der *Enter*-Taste wird der Inhalt des jeweils anderen Textfelds sofort aktualisiert.



Die Maßeinheit spielt bei der Programmierung keine Rolle, da sie für beide Eingabefelder identisch ist. Um beispielsweise einen Wasserbehälter mit 1 Kubikzentimeter Inhalt zu realisieren, ist eine Kugel mit dem Durchmesser von 1,241 Zentimetern erforderlich, für 1 Kubikmeter (1000 Liter) wären es 1,241 Meter:

3.9.3 Vom fetten zum schlanken Client

Lassen Sie sich durch den martialischen Titel nicht irritieren, wir wollen damit lediglich Ihr Interesse für eine Methodik wecken, die Ihnen hilft, den Horizont herkömmlicher Programmiertechniken zu überschreiten und damit einen leichteren Zugang zur OOP ermöglicht. Dabei gehen wir von folgender Erfahrung aus:

Typisch für den OOP-Ignoranten ist, dass er getreu der Devise "Hauptsache es funktioniert" mit Ausdauer und Beharrlichkeit immer wieder so genannte "fette" Clients (Fat Clients) programmiert. In einem solchen *Fat Client* ist in der Regel die gesamte Intelligenz (Geschäfts- bzw. Serverlogik) der Anwendung konzentriert, d h., eine Aufteilung in Klassen bzw. Schichten hat nie stattgefunden.

Ein qualifizierter objektorientierter Entwurf zeichnet sich aber dadurch aus, dass der Client möglichst "dumm" bzw. "dünn" ist. Ein *Thin Client* verwaltet ausschließlich das User-Interface, die Aufgaben beschränken sich auf die Entgegennahme der Benutzereingaben und deren Weiterleitung an die Geschäftslogik bzw. umgekehrt auf die Ausgabe und Anzeige der von der Geschäftslogik ermittelten Ergebnisse. Der Server hingegen umfasst die Geschäftslogik und kapselt die Intelligenz der Anwendung.

Die Vorteile einer solchen mehrschichtigen "Thin Client"-Strategie sind:

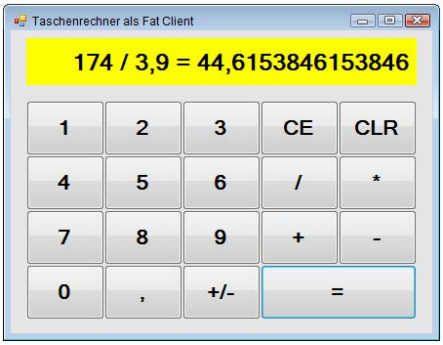
- gesteigerte Übersichtlichkeit und leichte Wiederverwendbarkeit der Software,
- Realisierung als verteilte Anwendung im Netzwerk ist möglich,
- Wartbarkeit und Erweiterbarkeit der Geschäftslogik sind möglich, ohne dass die Clients geändert werden müssten.

In unserem zweiteiligen Beispiel geht es um einen einfachen "Taschenrechner", den wir in zwei Versionen realisieren wollen.

In unserer ersten Windows Forms-Anwendung haben wir es mit einem Musterbeispiel für einen "fetten" Client zu tun. Im zweiten Teil verwandeln wir das Programm in eine mehrschichtige Anwendung mit einem "dünnen" Client. Neugierig geworden?

Oberfläche

So oder ähnlich sollte unser "Rechenkünstler" (egal ob dick oder dünn) zur Laufzeit aussehen:



Quellcode (Fat Client)

```
public partial class Form1 : Form
{   ...
```

Über die Bedeutung der folgenden drei globalen Zustandsvariablen brauchen wir wohl keine weiteren Worte zu verlieren:

```
    private string op;                        // aktueller Operator (+, - , *, /)
    private string reg1 = String.Empty;       // erstes Register   (Operand)
    private string reg2 = String.Empty;       // zweites Register  (Operand)
```

Wir wollen zur Steuerung des Programmablaufs eine spezielle Variable *state* verwenden, die den aktuellen Zustand speichert:

```
    private byte state = 1;           // aktuelles Register (1 oder 2)
```

Die nun folgenden Ereignisbehandlungen für die Schaltflächen des Ziffernblocks und die vier Rechenoperationen lassen wir diesmal nicht, wie ansonsten üblich, automatisch von der Entwicklungsumgebung generieren, sondern programmieren sie "per Hand". Grund dafür ist der geringere Schreibaufwand, da gemeinsame Eventhandler benutzt werden. Wir ergänzen also den Konstruktorcode mit dem Anmelden der entsprechenden Eventhandler (würden Sie das Anmelden wie üblich der IDE überlassen, so fänden Sie den automatisch generierten Code in der partiellen Klasse *Form1.Designer.cs*).

```
   public Form1()
   {
       InitializeComponent();
```

Die erste Gruppe bezieht sich auf den Ziffernblock und verweist auf die gemeinsam genutzte Methode *buttonZ_Click*:

```
       this.button1.Click += new EventHandler(this.buttonZ_Click);        // 1
       this.button2.Click += new EventHandler(this.buttonZ_Click);        // 2
       this.button3.Click += new EventHandler(this.buttonZ_Click);        // 3
       this.button4.Click += new EventHandler(this.buttonZ_Click);        // 4
       this.button5.Click += new EventHandler(this.buttonZ_Click);        // 5
       this.button6.Click += new EventHandler(this.buttonZ_Click);        // 6
       this.button7.Click += new EventHandler(this.buttonZ_Click);        // 7
       this.button8.Click += new EventHandler(this.buttonZ_Click);        // 8
       this.button9.Click += new EventHandler(this.buttonZ_Click);        // 9
       this.button10.Click += new EventHandler(this.buttonZ_Click);       // 0
       this.button11.Click += new EventHandler(this.buttonZ_Click);       // ,
```

Die zweite Gruppe bezieht sich auf die vier Operationstasten, welche auf die gemeinsame Methode *buttonOp_Click* verweisen:

```
       this.button12.Click += new EventHandler(this.buttonOp_Click);  // +
       this.button13.Click += new EventHandler(this.buttonOp_Click);  // -
       this.button14.Click += new EventHandler(this.buttonOp_Click);  // *
       this.button15.Click += new EventHandler(this.buttonOp_Click);  // :
   }
```

Typisch für die nun folgenden Implementierungen der Ereignisbehandlungen *buttonZ_Click* und *buttonOp_Click* ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind, die somit die Rolle einer "Programmweiche" übernimmt.

Zunächst die Zifferneingabe:

```
private void buttonZ_Click(object sender, EventArgs e)
{
    Button cmd = (Button)sender;
```

In Abhängigkeit von *state* wird die Eingabe zum ersten oder zum zweiten Register hinzugefügt. Der Wert der einzugebenden Ziffer wird dabei der *Text*-Eigenschaft des Buttons entnommen:

```
    switch (state)
    {
        case 1:                // zum ersten Operanden hinzufügen
            reg1 += cmd.Text[0];
            label1.Text = reg1; break;
        case 2:                // zum zweiten Operanden hinzufügen
            reg2 += cmd.Text[0];
            label1.Text = reg1 + " " + op + " " + reg2; break;
    }
}
```

Bei der Eingabe des Operators (+, -, *, /) wird ähnlich verfahren:

```
private void buttonOp_Click(object sender, EventArgs e)
{
    Button cmd = (Button) sender;
    switch (state)
    {
        case 1:                       // neuer Operand
            op = cmd.Text;
            state = 2; break;
        case 2:
            ergebnis();               // Zwischenergebnis mit altem Operand ermitteln
            op = cmd.Text; break;     // ... dann neuer Operand
    }
    label1.Text = reg1.ToString() + " " + op;
    reg2 = String.Empty;             // Register2 löschen
}
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis im Register1:

```
private void ergebnis()
{
    double r1 = Convert.ToDouble(reg1);
    double r2 = Convert.ToDouble(reg2);
    switch (op)
    {
        case "+":
```

```
                reg1 = (r1 + r2).ToString(); break;
            case "-":
                reg1 = (r1 - r2).ToString(); break;
            case "*":
                reg1 = (r1 * r2).ToString(); break;
            case "/":
                reg1 = (r1 / r2).ToString(); break;
        }
        reg2 = String.Empty;             // löscht Register2
    }
```

Die Ergebnistaste (=):

```
    private void buttonResult_Click(object sender, EventArgs e)
    {
        if (state == 2)
        {
            ergebnis();
            label1.Text += " = " + reg1;
            state = 1;
        }
        else
        {
            label1.Text = reg1;
            reg2 = "";            // löscht Register2
        }
    }
```

Letztes eingegebenes Zeichen löschen (CE):

```
    private void button17_Click(object sender, EventArgs e)
    {
        switch (state)
        {
            case 1:
                if (!(reg1 == ""))
                {
                    reg1 = reg1.Remove(reg1.Length - 1, 1);
                    label1.Text = reg1;
                }; break;
            case 2:
                if (!(reg2 == ""))
                {
                    reg2 = reg2.Remove(reg2.Length - 1, 1);
                    label1.Text = reg2;
                }; break;
        }
    }
```

Alle Register sowie Anzeige löschen und Anfangszustand herstellen (CLR):

```
    private void buttonCLR_Click(object sender, EventArgs e)
    {
        reg1 = String.Empty;
        reg2 = String.Empty;
        label1.Text = String.Empty;
        state = 1;
    }
```

Schließlich noch der Vorzeichenwechsel (+/-):

```
    private void buttonVZ_Click(object sender, EventArgs e)
    {
        double r;
        switch (state)
        {
            case 1:
                r = -Convert.ToDouble(reg1);
                reg1 = r.ToString();
                label1.Text = reg1; break;
            case 2:
                r = -Convert.ToDouble(reg2);
                reg2 = r.ToString();
                label1.Text = reg1 + " " + op + " " + reg2; break;
        }
    }
}
```

Test

Der Vorzug gegenüber üblichen Rechnern sticht sofort ins Auge: Man kann den Rechenvorgang mitverfolgen, weil der komplette Ausdruck angezeigt wird (siehe obige Abbildung).

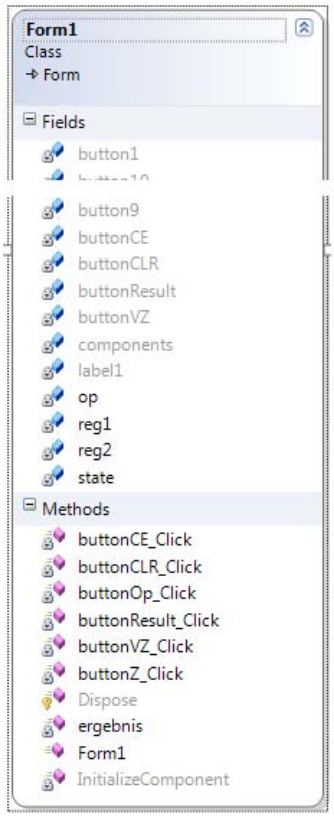
HINWEIS: Wie bei jedem anderen einfachen Taschenrechner auch, bleibt hier leider die Rangfolge der Operatoren (Punktrechnung geht vor Strichrechnung) unberücksichtigt. Bei der Eingabe von mehreren Operationen hintereinander, z.B. 3 + 4 * 12, ist deshalb zu beachten, dass erst die höherwertige Operation auszuführen ist (4 * 12).

Bemerkungen

- Den Programmablauf könnte man in Gestalt eines Zustandsüberführungsdiagramms (*State Chart*) noch anschaulicher darstellen, mehr dazu offenbart Kapitel 25 (UML).
- Leider ist die gesamte Intelligenz der Anwendung in der Benutzerschnittstelle *Form1* enthalten, also ein typischer "fetter" Client. Transparenz, Wiederverwendbarkeit und Wartbarkeit des Codes sind demzufolge katastrophal! Wie man das Programm auf ein höheres objektorientiertes Niveau heben kann, soll das folgende Beispiel zeigen.

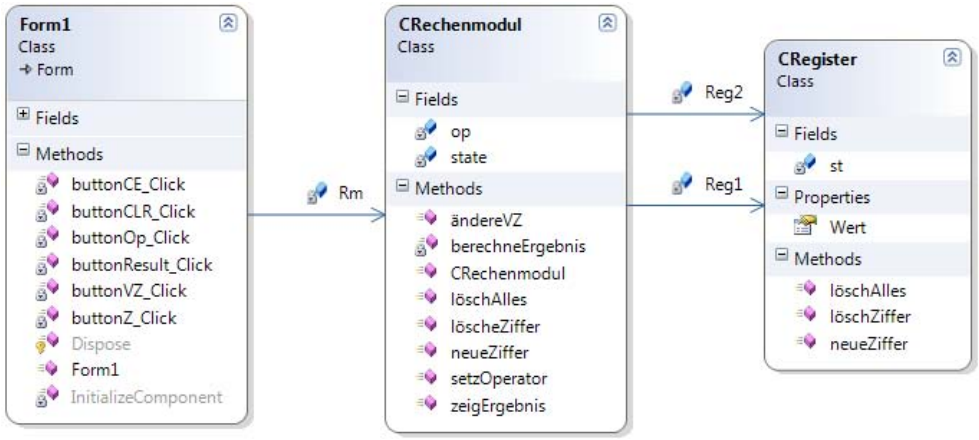
Abmagerungskur für den fetten Client

Dass es sich bei unserem Programm tatsächlich um einen Fat Client handelt, zeigt das zugehörige Klassendiagramm. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf *Form1.cs* und wählen Sie im Kontextmenü *Klassendiagramm anzeigen*. Es vergeht eine kleine Weile und dann bietet sich Ihnen der in der Abbildung nur auszugsweise gezeigt Anblick mit einer schier endlosen Auflistung der in der Klasse implementierten Felder und Methoden.



Schluss mit diesem Chaos! Durch Auslagern der Intelligenz in die Klassen *CRechenmodul* und *CRegister* erhalten wir nach einigem Hin und Her als Ergebnis der "Abmagerungskur" schließlich das abgebildete neue Klassendiagramm (siehe folgende Abbildung).

Die Klasse *Form1* (Thin Client) ist gegenüber dem Vorgänger deutlich abgemagert und beschränkt sich nur noch auf ihre eigentliche Aufgabe, die Verwaltung der Benutzerschnittstelle. Die Klassen *CRechenmodul* und *CRegister* stellen hingegen die zweischichtige Geschäftslogik (Server) der Anwendung dar, kapseln also die Intelligenz.



HINWEIS: Damit Sie im Klassendiagramm die durch Pfeilverbindungen gekennzeichneten Zuordnungen (Assoziationen) zwischen den Klassen sehen, müssen Sie mit der rechten Maustaste auf das Feld *Rm* in der Klasse *Form1* klicken und im Kontextmenü den Eintrag *Als Zuordnung anzeigen* wählen. Analog verfahren Sie mit den Feldern *Reg1* und *Reg2* in *CRechenmodul*.

Eine Erklärung des Klassendiagramms mit anderen Worten: Unser Thin Client benutzt eine Instanz der Klasse *CRechenmodul*. In dieser wiederum sind zwei Instanzen der Klasse *CRegister* enthalten. Hier geht es also noch nicht um Vererbung, Polymorphie etc, sondern nur um das zweckmäßige "Wegkapseln" von Funktionalität, wie das in unserem Fall etwa auch der physikalischen Realität entspricht, denn auch ein "richtiger" Taschenrechner enthält ein Rechenmodul, in diesem wiederum sind ein oder mehrere Register enthalten.

Allerdings stellt diese Thin Client-Lösung nur eine von mehreren Möglichkeiten dar und ist das Ergebnis einer Analyse des Ausgangscodes nach den Kriterien der Wiederverwendbarkeit ("Code Reuse").

HINWEIS: Mehr zu den erstaunlichen Fähigkeiten des in Visual Studio integrierten Klassendesigners erfahren Sie im Kapitel 25.

Quellcode für CRegister

```
public class CRegister
{
```

Die globale Variable *st* speichert den Registerinhalt als Zeichenkette:

```
    private string st = String.Empty;
```

Zugriff auf den numerischen Wert von *st:*

```
    public double Wert
    {
```

```
        get
        {
            try
                {return Convert.ToDouble(st);}
            catch
            {return 0;}
        }
        set
        {
            st = value.ToString();
        }
    }
```

Hinzufügen einer einzelnen Ziffer und Rückgabe des Anzeigestrings:

```
    public string neueZiffer(char z)
    {
        if ((Char.IsDigit(z)) || (z.ToString() == ","))
        {
             st += z;
             return (st);
        }
        else
            return "";
    }
```

Letzte Ziffer löschen und Anzeigestring zurückgeben:

```
    public string löschZiffer()
    {
        if (st.Length > 0) st = st.Remove(st.Length - 1, 1);
        return (st);
    }
```

Gesamtes Register löschen:

```
    public void löschAlles()
    {
        st = "";
    }
}
```

Quellcode für CRechenmodul

```
public class CRechenmodul
{
    private byte state = 1;                  // Startmodus (Zustandsvariable)
    private char op;                         // aktueller Operator
    private CRegister reg1, reg2;            // zwei Rechenregister
```

Im Konstruktor werden zwei Register-Objekte erzeugt:

```
public CRechenmodul()
{
    Reg1 = new CRegister();
    Reg2 = new CRegister();
}
```

Zifferneingabe in aktuelles Register:

```
public string neueZiffer(string ziff)
{
    char z = ziff[0];
    if (state == 1)             // zum ersten Register hinzufügen
        return (Reg1.neueZiffer(z));
    else                        // zum zweiten Register hinzufügen
        return (Reg1.Wert.ToString() + " " + op + " " + Reg2.neueZiffer(z));
}
```

Letzte Ziffer des aktuellen Registers löschen und resultierenden Registerinhalt zurückgeben:

```
public string löscheZiffer()
{
    if (state == 1)
    return (Reg1.löschZiffer());
    else
        return (Reg2.löschZiffer());
}
```

Vorzeichen des aktuellen Registers umkehren:

```
public string ändereVZ()
{
    if (state == 1)
    {
        Reg1.Wert = -Reg1.Wert;
        return (Reg1.Wert.ToString());
    }
    else
    {
        Reg2.Wert = -Reg2.Wert;
        return (Reg1.Wert.ToString() + " " + op + " " + Reg2.Wert.ToString());
    }
}
```

Der Operator wird übernommen. Rückgabewert ist der String des ersten Operanden mit abschließendem Operatorenzeichen:

```
public string setzOperator(string o)
{
    if (state == 1)
        state = 2;
    else
```

```
            berechneErgebnis();          // Zwischenergebnis (mit altem Operator) ermitteln
        op = o[0];                       // neuen Operator übernehmen
        Reg2.löschAlles();               // zweites Register löschen
        return (Reg1.Wert.ToString() + " " + op);
    }
```

Die abschließende Rechenoperation ausführen und das Ergebnis liefern:

```
    public string zeigErgebnis()
    {
        string s = "";
        if (state == 1)                  // im Startmodus wird noch nichts berechnet, ...
            Reg2.löschAlles();           // ... lediglich zweites Register wird gelöscht
        else
        {
            s = " = " + berechneErgebnis();
            state = 1;
        }
        return (s);
    }
```

Eine Hilfsmethode zum Ausführen der Rechenoperation nebst Abspeichern des Ergebnisses im ersten Register (überschreibt erstes Register mit Ergebnis der Operation):

```
    private string berechneErgebnis()
    {
        switch (op.ToString())
        {
             case "+":
                 Reg1.Wert = Reg1.Wert + Reg2.Wert; break;
             case "-":
                 Reg1.Wert = Reg1.Wert - Reg2.Wert; break;
             case "*":
                 Reg1.Wert = Reg1.Wert * Reg2.Wert; break;
             case "/":
                 Reg1.Wert = Reg1.Wert / Reg2.Wert; break;
        }
        Reg2.löschAlles();           // zweites Register löschen
        return (Reg1.Wert.ToString());
    }
```

Alle Register löschen und Startzustand wiederherstellen:

```
    public void löschAlles()
    {
        Reg1.löschAlles();
        Reg2.löschAlles();
        state = 1;
    }
}
```

Quellcode für Form1

Die Oberfläche unseres Thin Client entspricht 100%ig dem "fetten" Vorgänger. Die Programmierung ist allerdings – dank des *CRechenmodul*-Objekts – deutlich einfacher und transparenter geworden:

```
public partial class Form1 : Form
{
```

Einzige globale Variable[1] ist der Verweis auf die Klasse *CRechenmodul*:

```
    private CRechenmodul Rm;
```

Im Konstruktorcode müssen, wie im Vorgängerbeispiel, zunächst die Eventhandler für Ziffernblock und Operationstasten angemeldet werden:

```
    public Form1()
    {
        InitializeComponent();

        // Ziffernblock:
        this.button1.Click += new EventHandler(this.buttonZ_Click);  // 1
        ...
```

Neu ist das Instanziieren der Geschäftslogik (bzw. des Servers):

```
        Rm = new CRechenmodul();
    }
```

Eine Ziffer eingeben (0..9,):

```
    private void buttonZ_Click(object sender, EventArgs e)
    {
        Button cmd = (Button) sender;
        label1.Text = Rm.neueZiffer(cmd.Text);
```

Die Operation eingeben (+, -, *, /):

```
    private void buttonOp_Click(object sender, EventArgs e)
    {
        Button cmd = (Button) sender;
        label1.Text = Rm.setzOperator(cmd.Text);
    }
```

Ergebnis anzeigen (=):

```
    private void buttonResult_Click(object sender, EventArgs e)
    {
          label1.Text += Rm.zeigErgebnis();
    }
```

[1] Ok, der OOP-Purist wird hier ein wenig die Nase rümpfen, denn ein ideales UI ist zustandslos (siehe Kapitel 26, Observer Pattern).

Letztes eingegebenes Zeichen löschen (CE):

```
    private void buttonCE_Click(object sender, EventArgs e)
    {
        label1.Text = Rm.löscheZiffer();
    }
```

Alle Register sowie Anzeige löschen und Anfangszustand wieder herstellen:

```
    private void buttonCLR_Click(object sender, EventArgs e)
    {
        Rm.löschAlles();
        label1.Text = "";
    }
```

Vorzeichenwechsel (+/-):

```
    private void buttonVZ_Click(object sender, EventArgs e)
    {
        label1.Text = Rm.ändereVZ();
    }
}
```

Test

Sie werden keinerlei Unterschied in Aussehen und Verhalten unseres "dünnen"- Taschenrechners zu seinem "fetten" Vorgänger feststellen, was uns in der Auffassung bestätigt, dass den Hauptnutzen aus der OOP nicht der Anwender, sondern der Programmierer hat!

Bemerkungen

- Unter Einsatz einer Formelparser-Klasse wären auch Klammerrechnungen möglich, das dürfte weniger aufwändig sein als das Hinzufügen weiterer Register.
- Einen wesentlich leistungsfähigeren wissenschaftlichen Taschenrechner, der allerdings nach einem völlig anderen Prinzip funktioniert (Code DOM), finden Sie in unserem [Visual C# 2012 Kochbuch].

3.9.4 Schnittstellenvererbung verstehen

Ein mächtiges Feature der OOP ist das Konzept der Schnittstellenvererbung (siehe Abschnitt 3.8). Die Schnittstellenvererbung erschließt sich dem Einsteiger am leichtesten, wenn er sich vorher das Konzept abstrakter Klassen und Methoden (siehe Abschnitt 3.7.1) verinnerlicht hat.

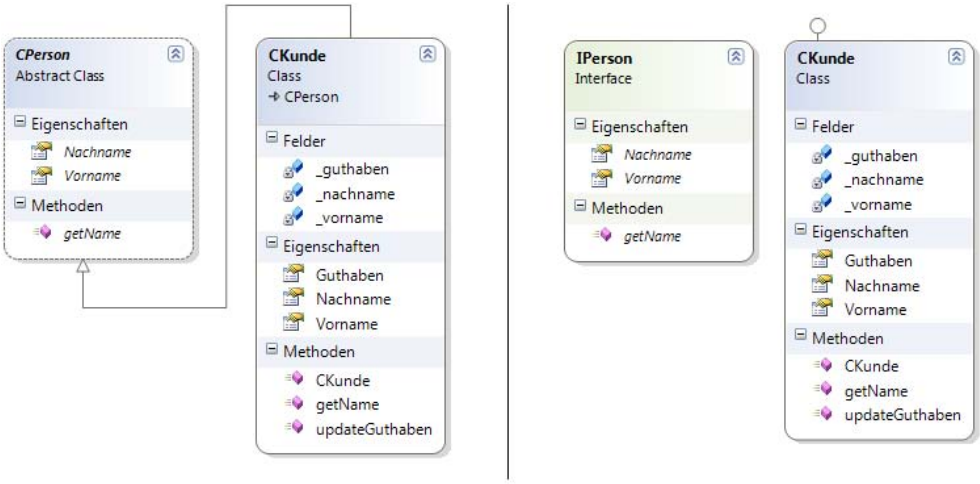
In unserem kleinen Demobeispiel wollen wir Geldbeträge auf das Konto eines Kunden einzahlen bzw. von dort abheben. Die erste Lösung soll mittels einer abstrakten Klasse, die nur über abstrakte Methoden verfügt, erfolgen[1]. Die zweite Lösung benutzt eine Schnittstelle (Interface).

[1] Dies ist allerdings die extremste Form der Implementierungsvererbung, denn es wird de facto keinerlei Code vererbt.

Klassendiagramme

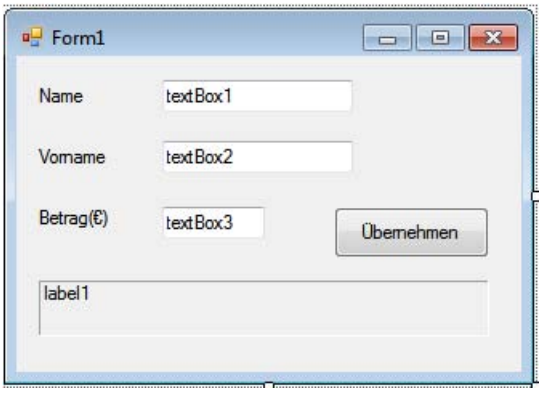
Die erste Variante zeigt das links abgebildete Klassendiagramm. Hier erbt die Klasse *CKunde* von der abstrakten Klasse *CPerson*. Letztere verfügt ausschließlich über abstrakte Klassenmitglieder. Diese enthalten nur die Eigenschafts- bzw. Methodendeklaration, also keinerlei Code. Die Implementierung muss komplett in der erbenden Klasse *CKunde* erfolgen.

Das rechte Klassendiagramm zeigt die zweite Lösung, bei welcher die abstrakte Klasse *CPerson* von der Schnittstelle *IPerson* ersetzt wird. Weitere Unterschiede sind auf den ersten Blick nicht zu erkennen, dazu müssen wir uns den Quellcode näher anschauen.



Oberfläche Form1

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie das Startformular wie gezeigt.



Beginnen wir mit der ersten Variante!

Quellcode CPerson

Fügen Sie dem Projekt eine neue Klasse *CPerson* zu:

```
public abstract class CPerson
{
    public abstract string Nachname
    { get; set; }

    public abstract string Vorname
    { get; set; }

    public abstract string getName();
}
```

Wie Sie sehen, ist die Klasse abstrakt und enthält die abstrakten Eigenschaften *Nachname* und *Vorname*) sowie die abstrakte Methode *getName*.

Quellcode CKunde

Fügen Sie dem Projekt eine Klasse *CKunde* hinzu:

```
class CKunde : CPerson
{
        private string _nachname;
        private string _vorname;
        private decimal _guthaben;
```

Die von *CPerson* geerbten abstrakten Klassenmitglieder müssen überschrieben werden:

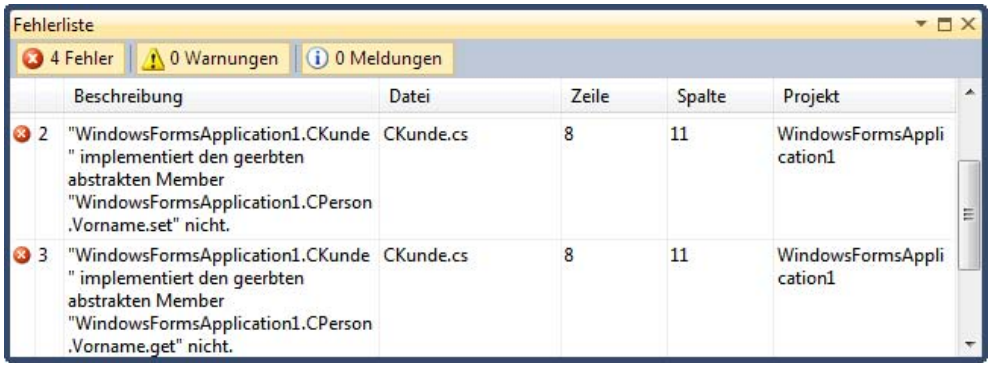
```
        public override string Nachname
        {
            get { return _nachname; }
            set {_nachname = value; }
        }

        public override string Vorname
        {
            get { return _vorname; }
            set { _vorname = value; }
        }

        public override string  getName()
        {
         return _vorname + " " +_nachname ;
        }
```

Es folgen die normalen Klassenmitglieder:

```
        public CKunde(string vor, string nach)       // ein Konstruktor
        {
```

```
            _vorname = vor; _nachname = nach;
        }

        public decimal Guthaben
        {
            get { return _guthaben; }
        }

        public decimal updateGuthaben(decimal betrag)
        {
            return _guthaben += betrag;
        }
    }
```

HINWEIS: Es müssen **alle** geerbten abstrakten Klassenmitglieder überschrieben werden, ansonsten erfolgt eine Fehlermeldung des Compilers (siehe Abbildung).



Quellcode Form1

```
...
public partial class Form1 : Form
{
```

Zu Beginn wird ein Kunde erzeugt, initialisiert und angezeigt:

```
    CKunde kunde1 = new CKunde("Max", "Müller");

    private void Form1_Load(object sender, EventArgs e)
    {
        textBox1.Text = kunde1.Vorname;
        textBox2.Text = kunde1.Nachname;
        textBox3.Text = "0";
    }
```

Bei jedem Klick auf die Schaltfläche werden Vor- und Nachname des Kunden neu zugewiesen. Der eingegebene Betrag wird dem Guthaben hinzugefügt bzw. (bei negativem VZ) abgezogen:

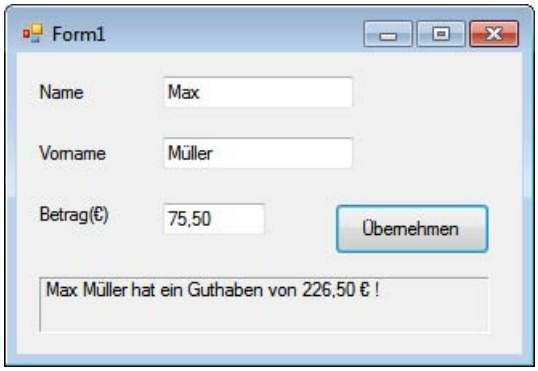
```
    private void button1_Click(object sender, EventArgs e)
    {
        kunde1.Vorname = textBox1.Text;
        kunde1.Nachname = textBox2.Text;
        decimal betrag = Convert.ToDecimal(textBox3.Text);
        kunde1.updateGuthaben(betrag);
```

Die abgeschlossene Buchung wird mit einem Meldungstext quittiert:

```
        label1.Text = kunde1.getName() + " hat ein Guthaben von " +
                                                kunde1.Guthaben.ToString("C") + " !";
    }
}
```

Test

Nehmen Sie einige Ein- oder Auszahlungen vor.



Nach erfolgreichem Test dieser ersten Variante wollen wir die zweite Variante in Angriff nehmen und die Klasse *CPerson* durch ein Interface *IPerson* ersetzen.

Quellcode IPerson

Benennen Sie einfach im Projektmappenexplorer die Klasse *CPerson* in *IPerson* um (die Abfrage, ob auch alle Verweise geändert werden sollen, beantworten Sie positiv):

```
public interface IPerson
{
    string Nachname
    { get; set; }

    string Vorname
    { get; set; }

    string getName();
}
```

Quellcode CKunde

Auch hier sind nur minimale Änderungen erforderlich, denn bei den von *IPerson* geerbten Schnittstellenmitgliedern fehlen lediglich die *override*-Modifizierer.

```
class CKunde : IPerson
{
    ...
     public string Nachname                  // ohne override !
     {
          get { return _nachname; }
          set {_nachname = value; }
     }
     usw. ...
}
```

Das war es auch schon, denn der Quellcode von *Form1* kann unverändert bleiben!

Beim Testen des Codes werden Sie keinerlei Veränderungen zum Vorgängerprojekt feststellen, allerdings dürfte der gesamte Code etwas übersichtlicher geworden sein.

Bemerkungen

- Ebenso wie von abstrakten Klassen können auch von Schnittstellen keine Instanzen erzeugt werden.
- Es müssen ausnahmslos **alle** Schnittstellenmitglieder implementiert werden (eine Schnittstelle ist wie ein Vertrag, der bedingungslos einzuhalten ist!).
- Zugriffsmodifizierer (*public*) für die Mitglieder einer Schnittstelle fehlen komplett, denn diese haben in einer Schnittstellendefinition nichts zu suchen.
- Eine Klasse kann von mehreren Schnittstellen erben, jedoch nur von einer einzigen Klasse. Das ist der Hauptunterschied (oder auch Hauptvorteil!) der Schnittstellenvererbung gegenüber der Implementierungsvererbung.

Kapitel 4

Arrays, Strings, Funktionen

Ausgerüstet mit den in Kapitel 3 vermittelten Grundkenntnissen der objektorientierten Programmierung wollen wir nun die sprachlichen Grundlagen von C# weiter ausbauen.

Wir werden Ihnen im vorliegenden Kapitel weiteres Handwerkszeug, wie den objektorientierten Umgang mit Arrays, Zeichenketten, regulären Ausdrücken, Zahlenformatierungen, mathematischen und Datumsfunktionen, vermitteln.

4.1 Datenfelder (Arrays)

Arrays eignen sich zum Speichern mehrerer Variablen eines bestimmten Datentyps. Gegenüber der Verwendung vieler einzelner "normaler" Variablen ergibt sich der Vorteil, dass nur ein einziger Variablenname benötigt wird, die Unterscheidung erfolgt über einen Index.

HINWEIS: Der untere Index eines Arrays ist standardmäßig null. Der höchste Index ist deshalb immer um Eins niedriger als die Länge des Arrays.

Da auch Arrays zu den Referenz- bzw. Verweistypen zählen, geht es im Unterschied zu den Wertetypen nicht mehr ohne ein Minimum an objektorientiertem Vokabular. Wie jedes andere Objekt wird auch eine Array-Variable in drei Etappen zum Leben erweckt:

- Deklarieren
- Erzeugen
- Initialisieren

Im vorliegenden Abschnitt wollen uns zunächst nur auf die für den Einstieg wichtigsten Grundlagen beschränken.

4.1.1 Array deklarieren

Sie deklarieren ein Array wie eine normale Variable, lediglich nach dem Datentypbezeichner folgt ein eckiges Klammerpärchen [].

SYNTAX: *Datentyp*[] *arrVariable*;

Der Datentyp des Arrays kann beliebig sein.

BEISPIEL 4.1: Eine *int*-Variable *a* wird deklariert

C#
```
int[] a;
```

Die Größe des Arrays wird bei der Deklaration nicht angegeben, denn es existiert noch keine Instanz dieses Objekts, lediglich ein Verweis (Referenz) ins Leere (*null*) wird auf dem Stack angelegt.

4.1.2 Array instanziieren

Eine Array-Instanz erzeugen Sie mit dem Schlüsselwort *new*:

SYNTAX: *arrVariable* = new *Datentyp*[*Dimension*];

BEISPIEL 4.2: Die im Vorgängerbeispiel deklarierte *Array*-Variable wird mit vier Elementen erzeugt

C#
```
a = new int[4];
```

Die Array-Variable *a* zeigt nun nicht mehr ins Leere, sondern auf einen konkreten Speicherbereich im Heap, der vier Integer aufnehmen kann.

Sie können auch das Deklarieren und Instanziieren innerhalb einer einzigen Anweisung erledigen.

SYNTAX: *Datentyp*[] *arrVariable* = new *Datentyp*[*Dimension*];

BEISPIEL 4.3: Eine Zusammenfassung der beiden Vorgängerbeispiele

C#
```
int[] a = new int[4];
```

4.1.3 Array initialisieren

Arrays werden beim Erzeugen mit Standardwerten initialisiert (einfache Datentypen mit *0*, boolesche Elemente mit *false*, Verweistypen mit *null*). Um andere Werte zuzuweisen, müssen Sie diese in geschweiften Klammern { }, getrennt durch Kommata, auflisten und an die *new*-Anweisung anhängen.

SYNTAX: *arrVariable* = new *Datentyp*[*Dimension*] {wert1, wert2, ... };

BEISPIEL 4.4: Die *int*-Array-Variable *a* wird deklariert und mit Werten gefüllt.

C#
```
int[] a;                          // Deklarieren
a = new int[4] {10, 9, 8, 7};     // Erzeugen und Initialisieren
```

Wem das zu umständlich ist, der kann auch von einer Kurzform Gebrauch machen, die ohne den *new*-Operator auskommt und bei der die drei Schritte Deklarieren, Erzeugen und Initialisieren in einer einzigen Anweisung zusammengefasst sind. Die Angabe der Indexgrenze entfällt, denn diese ergibt sich ja automatisch aus der Anzahl der übergebenen Werte.

SYNTAX: *Datentyp*`[]` *arrVariable* = {wert1, wert2, ... };

BEISPIEL 4.5: Die Kurzform des Vorgängerbeispiels

C#
```
int[] a = {10, 9, 8, 7};            // Deklarieren, Erzeugen und Initialisieren
```

BEISPIEL 4.6: Ein *string*-Array mit fünf initialisierten Feldern

C#
```
string[] arbeitsTage = {"Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag"};
```

Die Werte, mit denen das Array initialisiert wird, müssen nicht immer Konstanten bzw. Literale sein, sondern können natürlich auch erst zur Laufzeit berechnet werden, wie das folgende etwas anspruchsvollere Beispiel beweist.

BEISPIEL 4.7: Das *int*-Array *a* wird mit vier Zufallszahlen initialisiert.

C#
```
Random z = new Random();              // Objektvariable der Random-Klasse erzeugen
int[] a = new int[4] {z.Next(1,7), z.Next(1,7), z.Next(1,7), z.Next(1,7)};
```

Die *Random*-Klasse stellt einen Generator für Pseudozufallszahlen dar. Die *Next*-Methode liefert hier Zufallszahlen, die zwischen 1 und 6 liegen (Würfelprinzip).

4.1.4 Zugriff auf Array-Elemente

Der Lese-/Schreibzugriff auf einzelne Elemente des Arrays erfolgt wie bei normalen Variablen, nur dass dem Namen der Array-Variablen der in (eckigen) Klammern stehende Index folgt.

HINWEIS: Da die Indizierung eines Arrays bei 0 beginnt, hat das erste Element den Index 0. Mit dem Index 1 greifen Sie auf das zweite Element zu.

BEISPIEL 4.8: Das *int*-Array *a* wird mit neuen Werten gefüllt

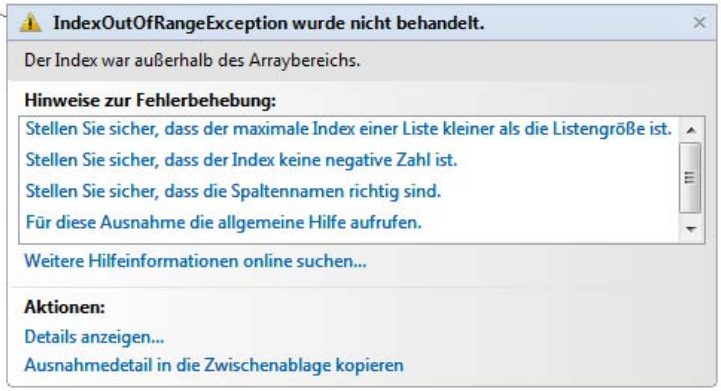
C#
```
int[] a = new int[4];
a[0] = 100; a[1] = 33; a[2] = 80; a[3] = 75;
```

Indexüberprüfung

Beim Zugriff auf die Array-Elemente (also erst zur Laufzeit) wird geprüft, ob der Index im erlaubten Bereich liegt.

BEISPIEL 4.9: *(Fortsetzung)* Obiges Array hat vier Elemente, die mit 0, 1, 2, 3 indiziert sind, der Zugriff auf den Index 4 erzeugt einen Laufzeitfehler

```
int[] a = new int[4];
a[0] = 100; a[1] = 33; a[2] = 80; a[3] = 75;
a[4] = 50;
```



Oberen Array-Index ermitteln

Sie wissen mittlerweile, dass der untere Index eines Arrays immer 0 ist. Was aber, wenn der obere Index unbekannt ist? In diesem Fall fragen Sie einfach die *Length*-Eigenschaft des Arrays ab, welche die Anzahl der Feldelemente zurückgibt. Verringern Sie diese um eins und Sie haben den oberen Feldindex.

BEISPIEL 4.10: In das letzte Element des Arrays *a* wird ein neuer Wert geschrieben

C#
```
int max = a.Length - 1;
a[max] = 100;
```

4.1.5 Zugriff mittels Schleife

Für den sequenziellen Zugriff auf alle Array-Elemente eignen sich prinzipiell alle unter 2.5.2 angegebenen Schleifenanweisungen.

for-Schleife

BEISPIEL 4.11: Ein *int*-Array wird mit den Zahlen 0 ... 99 gefüllt

C#
```
int[] a =  new int[100];
for (int i = 0; i != a.Length; i++)
  a[i] = i;
```

foreach-Schleife

Eine weitere wichtige Zugriffsart ist die uns bislang unbekannte *foreach*-Schleife, mit welcher alle Elemente des Arrays durchlaufen werden können, ohne dass deren Anzahl bekannt sein muss.

SYNTAX:
```
foreach (element in array)
{
  Anweisungen;
  [break];
};
```

Wie jede andere Schleife kann auch die *foreach*-Schleife mit dem optionalen *break* vorzeitig verlassen werden.

BEISPIEL 4.12: Das im Vorgängerbeispiel gefüllte Array wird ausgelesen

C#
```
foreach (int feld in a)
{
   MessageBox.Show(feld.ToString());      // zeigt nacheinander 0, 1, 2 ... 10
   if (feld == 10) break;
}
```

Leider ist mit einer *foreach*-Schleife nur der Lesezugriff auf die Feldelemente möglich.

BEISPIEL 4.13: Das Füllen eines Arrays misslingt

C#
```
int[] a =  new int[100];
int i = 0;
foreach (int feld in a)
  feld = i++;                 // Fehler, weil feld schreibgeschützt ist
```

HINWEIS: In der Regel ist das Durchlaufen eines Arrays mit *for*-Schleife schneller als mit *foreach*-Schleife.

Eine besonders wichtige Rolle spielt die *foreach*-Schleife im Zusammenhang mit dem Durchlaufen von Objektauflistungen (siehe Kapitel 5, *IEnumerable*-Interface).

4.1.6 Mehrdimensionale Arrays

Ein Array darf auch mehr als nur eine einzige Dimension haben. Die einzelnen Dimensionen werden bei Deklaration bzw. Zugriff durch Komma voneinander getrennt.

SYNTAX: *Datentyp*[, ,] *arrVariable* = new *Datentyp*[*Dimension1, Dimension2, Dimension3*];

Sie können sich beispielsweise ein zweidimensionales Array als "Regal" vorstellen, welches nach seiner Definition zunächst aus "leeren" Fächern besteht, die in mehreren Reihen übereinander angeordnet sind und die jeweils eine Variable des deklarierten Datentyps aufnehmen können[1].

[1] Ein bekanntes Einsatzgebiet für mehrdimensionale Arrays ist z.B. die Matrizenrechnung in der Mathematik.

BEISPIEL 4.14: Ein (zweidimensionales) Gleitkomma-Array mit 10 Zeilen und 20 Spalten wird deklariert.

C#
```
float[, ] c = new float[10, 20];    //  c(0,0) ... c(9, 19)
```

Mit den folgenden Anweisungen speichern Sie in jedem Feld den Wert *5.5*:

```
for (int i = 0; i != 10; i++)         // für alle Zeilen
{
  for (int j = 0; j != 20; j++)            // für alle Spalten
  {
    c[i,j] = 5.5F;
  }
}
```

HINWEIS: Für den reinen Lesezugriff ist auch die Verwendung einer *foreach*-Schleife möglich. In diesem Fall benötigt man – wie bei eindimensionalen Arrays – nur eine Schleife, die dann von links oben nach rechts unten durchlaufen wird.

BEISPIEL 4.15: Der Inhalt des im Vorgängerbeispiel erzeugten Arrays wird in einer *ListBox* angezeigt.

C#
```
...
foreach (float f in c)
{
    listBox1.Items.Add(f.ToString());
}
```

Dimensionsgrenzen ermitteln

Zur Bestimmung der Länge der einzelnen Dimensionen eines mehrdimensionalen Arrays ist die *Length*-Eigenschaft ungeeignet. Verwenden Sie stattdessen die *GetLength*- oder *GetUpperBound*-Methode der *Array*-Klasse (siehe Abschnitt 4.1.10).

BEISPIEL 4.16: Das Vorgängerbeispiel kann auch eleganter programmiert werden.

C#
```
float[, ] c = new float[10, 20];
for (int i = 0; i < c.GetLength(0); i++)         // für erste Dimension
{
  for (int j = 0; j < c.GetLength(1); j++)            // für zweite Dimension
  {
    c[i,j] = 5.5F;
  } }
```

Mehrdimensional initialisieren

Auch mehrdimensionale Arrays lassen sich mit Anfangswerten initialisieren, die Werte der einzelnen Dimensionen werden in geschweiften Klammerpaaren hintereinander aufgeführt.

BEISPIEL 4.17: Ein zweidimensionales *String*-Array wird erzeugt und mit Namen und Wohnort von drei Personen initialisiert.

C#

```
string[,] personen = {{"Meier", "Berlin"}, {"Schultze", "Leipzig"}, {"Krause", "Bonn"}};
string s = personen[2,1];
MessageBox.Show(s);            // zeigt "Bonn"
```

Ergebnis

	0	1	2
0	Meier	Schultze	Krause
1	Berlin	Leipzig	Bonn

4.1.7 Zuweisen von Arrays

Ebenso wie Sie den Inhalt einer Variablen einer anderen Variablen zuweisen können, z.B. a = b, können Sie auch einem Array ein anderes Array zuweisen. Da aber Arrays in allen .NET-Programmiersprachen *Referenztypen* sind, speichern Array-Variablen also direkt keine Werte, sondern nur Referenzen (Zeiger) auf einen bestimmten Speicherbereich. Normale Datentypen hingegen zählen bekanntlich zu den *Wertetypen*.

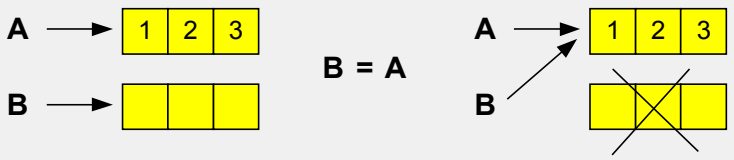
Das gegenseitige Zuweisen von Array-Variablen führt also nicht zu einem Kopieren der Werte, sondern lediglich zum "Verbiegen" eines Zeigers, d h. zum Zuweisen einer neuen Speicheradresse.

BEISPIEL 4.18: Ein mit den Werten 1, 2 und 3 initialisiertes Array *a* wird einem Array *b* zugewiesen, anschließend wird im *b*-Array ein Feld geändert.

C#

```
int[] a = {1, 2, 3};
int[] b = new int[a.Length];
b = a;                                  // kein Kopieren, sondern nur Referenzieren!
b[0] = 5;
MessageBox.Show(b[0].ToString());       // zeigt erwartungsgemäß "5" an
MessageBox.Show(a[0].ToString());       // zeigt nicht "1", sondern ebenfalls "5" an!
```

Im Beispiel wurde der Zeiger, der vorher das Array *b* referenzierte, "verbogen" und zeigt nun ebenfalls auf *a*. Was aber passiert mit dem Speicherbereich des "in der Luft hängenden" *b*-Arrays? Er steht nicht mehr zur Verfügung und wird bei nächster Gelegenheit vom Garbage Collector entsorgt (Abbildung).



Um ein Array wirklich in ein anderes zu kopieren, können Sie zunächst einen umständlichen Weg beschreiten und die Werte einzeln zuweisen.

BEISPIEL 4.19: Das Vorgängerbeispiel wird so geändert, dass ein echtes Kopieren stattfindet.

C#
```
int[] a = {1, 2, 3};
int[] b = new int[a.Length];
for (int i = 0; i < a.Length; i++)
  b[i] = a[i];
b[0] = 5;
MessageBox.Show(a[0].ToString());      // zeigt "1" an
MessageBox.Show(b[0].ToString());      // zeigt "5" an
```

HINWEIS: Eine elegantere Lösung mittels *Clone*-Methode finden Sie im Abschnitt 4.1.10.

4.1.8 Arrays aus Strukturvariablen

Ist für den Einsteiger der Umgang mit Strukturvariablen schon gewöhnungsbedürftig, so ist deren Einbau in Arrays für manchen eine schier unüberwindliche Barriere. Dabei ist das gar nicht so sehr schwierig zu verstehen, wenn man ein plausibles Beispiel parat hat.

BEISPIEL 4.20: Spieler einer Fußballmannschaft

C#
```
struct Spieler
{
  public string name;
  public int tore;
  public decimal gehalt;
  public bool auswahlSpieler;
}
```

Deklaration und Zugriff:

```
Spieler[] alleSpieler = new Spieler[10];    // eine Mannschaft = 11 Spieler
alleSpieler[0].name = "Neuer";
alleSpieler[3].name = "Schweinsteiger";
```

BEISPIEL 4.21: Eine komplette Fußballmannschaft

C#
```
struct Mannschaft
{
  public string verein;
  public Spieler[] alleSpieler;           // Verweis auf Substruktur!
  public string trainer;
}
```

Deklaration und Zugriff:

```
Mannschaft mannschaft1;                  // Mannschafts-Variable
mannschaft1.alleSpieler = new Spieler[10];
mannschaft1.verein = "Bayern München";
mannschaft1.alleSpieler[0].name = "Neuer";
```

BEISPIEL 4.21: Eine komplette Fußballmannschaft

```
mannschaft1.alleSpieler[3].tore = 12;
mannschaft1.trainer.name = "Heinke";
```

BEISPIEL 4.22: Alle Spieler mit mehr als zehn Toren werden zu Auswahlspielern.

C#
```
for (int i = 0; i < mannschaft1.alleSpieler.Length; i++)
 if (mannschaft1.alleSpieler[i].tore > 10)
    mannschaft1.alleSpieler[i].auswahlSpieler = true;
```

Arrays aus Strukturvariablen sind die universellere Alternative zu *verzweigten Arrays*, bei denen die Array-Elemente selbst wieder Arrays sind und auf die wir hier aus Platzgründen nicht näher eingehen wollen.

4.1.9 Löschen und Umdimensionieren von Arrays

Felder löschen

Um die Felder eines Arrays zu löschen (sie werden nicht entfernt!), verwenden Sie die *Clear*-Methode der *Array*-Klasse. Es handelt sich hierbei um eine statische Methode der *Array*-Klasse (siehe Abschnitt 4.1.10).

SYNTAX: `Array.Clear(arr, index, anz);`

arr = die Array-Variable; *index* = Startindex; *anz* = Anzahl der zu löschenden Felder

Die Felder werden auf ihre Standardwerte zurückgesetzt (*0* bei Wertetypen, *null* bei Verweistypen, *false* bei booleschen Werten).

BEISPIEL 4.23: Das vierte Element eines Arrays (Index = 3) wird gelöscht.

C#
```
int[] a = {5, 6, 7, 8, 9};      // Kurzform zum Erzeugen und Initialisieren eines Arrays
MessageBox.Show(a[3].ToString());    // viertes Element hat Wert 8
Array.Clear(a,3,1);                  // viertes Element wird gelöscht
MessageBox.Show(a[3].ToString());    // viertes Element hat Wert 0
```

Array löschen

Wollen Sie das komplette Array entfernen, müssen Sie der Array-Variablen den Wer *null* zuweisen (dereferenzieren). Diese zeigt dann gewissermaßen ins Leere und nicht mehr auf den entsprechenden Heap-Speicherbereich, in welchem das Array abgelegt ist. Das herrenlose Array wird allerdings nicht sofort, sondern erst bei nächstbester Gelegenheit vom .NET-Garbage Collector entsorgt.

BEISPIEL 4.24: Array löschen

C# Eine Array-Variable wird deklariert, initialisiert und dereferenziert. Der folgende Zugriff auf ein beliebiges Element führt danach zu einem Fehler.

```
int[] a = new int[4] {5, 6, 7, 8};   // Deklarieren und Initialisieren
a = null;            // Dereferenzieren
a[0]= 2;             // Fehler
```

Statt die Array-Variable *null* zu setzen, kann sie auch mit einem neuen Array quasi "überschrieben" werden.

BEISPIEL 4.25: Die Array-Variable *a* zeigt zunächst auf ein Array mit vier und danach auf ein neues Array mit zwei Feldern.

C#
```
int[] a= new int[4] {5, 6, 7, 8};     // initialisiertes Array mit vier Feldern
a = new int[2] {13, 14};              // neues initialisiertes Array mit zwei Feldern
```

Array-Dimensionen ändern

Im Vorgängerbeispiel haben Sie bereits eine Möglichkeit kennen gelernt, um die Abmessungen eines Arrays zur Laufzeit zu ändern. Allerdings geht dabei der alte Array-Inhalt verloren.

Wenn die Größe des Arrays nicht konstant bleibt, sondern sich während der Laufzeit ändert, oder wenn Sie Array-Elemente hinzufügen oder löschen wollen, kommen Sie nicht um einigen Programmieraufwand herum[1], wobei Sie meistens zwischen einem Quell- und einem Ziel-Array hin- und herkopieren müssen.

Allerdings ist dieser Aufwand nur für die von uns bislang betrachteten Arrays der *System.Array*-Klasse erforderlich. Wesentlich komfortablere Möglichkeiten zum Hinzufügen und Löschen bietet die Klasse *ArrayList* oder die generische *List* (siehe Kapitel 5).

4.1.10 Eigenschaften und Methoden von Arrays

Bereits mitten drin im OOP-Geschehen kommen Sie jetzt nicht umhin, sich auch mit den *Eigenschaften* und *Methoden* von Arrays anzufreunden. Lassen Sie sich nicht verwirren, wenn im Folgenden von zwei verschiedenen Arten von Eigenschaften/Methoden die Rede ist:

- *Instanzeigenschaften und -methoden*
  Diese wenden Sie auf eine Array-**Instanz**, also auf eine Array-Variable an.
- *Klasseneigenschaften und -methoden* (auch als *statische* Eigenschaften/Methoden bekannt)
  Diese wenden Sie direkt auf die ***Array*-Klasse** an.

Wenn Sie sich die Beispiele dieses Abschnitts näher anschauen, dürfte Ihnen der Unterschied bald klar werden. Die folgenden Tabellen sollen lediglich einen Überblick vermitteln[2].

---

[1] In Visual Basic gibt es dafür die *ReDim*-Anweisung für die so genannten *dynamischen* Arrays.

[2] Die exakte Syntax entnehmen Sie bitte der Dokumentation.

Wichtige Eigenschaften und Methoden von Array-Variablen

Member	Beschreibung
`Length`	... liefert Gesamtanzahl der Array-Elemente
`Rank`	... liefert die Anzahl der Array-Dimensionen
`Clone(...)`	... kopiert alle Elemente aus Quell- in Ziel-Array
`CopyTo(...)`	... kopiert Elemente ab *index* von Quell- in Ziel-Array
`GetLength`	... liefert die Anzahl Elemente einer bestimmten Dimension
`Initialize(...)`	... initialisiert alle Array-Elemente auf Standardwerte

Ein Möglichkeit zum 1:1-Kopieren von Arrays bietet die *Clone*-Methode. Die Abmessungen des Ziel-Arrays müssen dabei **nicht** unbedingt mit denen des Quell-Arrays übereinstimmen.

BEISPIEL 4.26: Der Inhalt des Arrays *a* wird in das Array *b* kopiert

C#

```
int[] a = {1, 2, 3};        // Quell-Array
int[] b;                    // Ziel-Array
```

Da die *Clone*-Methode einen Wert vom *object*-Datentyp zurückgibt, musste eine explizite Typumwandlung in ein *int[]*-Array vorgenommen werden:

```
b = (int[]) a.Clone();      // Typecasting
b[0] = 5;
MessageBox.Show(a[0].ToString());       // zeigt "1" an
MessageBox.Show(b[0].ToString());       // zeigt "5" an
```

Wichtige Methoden der Array-Klasse

Methode	Beschreibung
`BinarySearch(...)`	... dient zur (schnellen) binären Suche in einem sortierten (!) Array
`Clear(...)`	... löscht die Werte von Elementen
`IndexOf(...)`	... dient zur (langsamen) sequenziellen Suche in einem Array
`Sort(...)`	... dient zur vollständigen bzw. teilweisen Sortierung eines Arrays

BEISPIEL 4.27: Ein eindimensionales *int*-Array mit zehn Elementen wird nach der Zahl 11 durchsucht. Vorher muss es sortiert werden, um *BinarySearch* anwenden zu können.

C#

```
int[] a = {13, 11, 5, 3, 12, 1, 3, 6, 4, 2};
Array.Sort(a);                // neue Reihenfolge {1, 2, 3, 3, 4, 5, 6, 11, 12, 13}
int i;
for (i = 0; i < a.Length; i++)
   MessageBox.Show(a[i].ToString());   // zeigt sortierte Reihenfolge an
i = Array.BinarySearch(a, 11);         // Suche nach der Zahl 11
```

BEISPIEL 4.27: Ein eindimensionales *int*-Array mit zehn Elementen wird nach der Zahl 11 durchsucht. Vorher muss es sortiert werden, um *BinarySearch* anwenden zu können.

```
if (i > 0)
   MessageBox.Show("11 an Position " + i.ToString() + " gefunden!");
else
   MessageBox.Show("Nichts gefunden!");
```

4.1.11 Übergabe von Arrays

Von den vielen Möglichkeiten zur Parameter- und Ergebnisübergabe, die es neben den einfachen Datentypen gibt (eigentlich können Sie beliebige Objekte, wie z.B. auch *Collections,* übergeben), wollen wir uns den Umgang mit Arrays näher anschauen.

Arrays als Parameter

An Methoden können auch Arrays übergeben werden. Dazu deklarieren Sie diese mit leeren Klammern.

BEISPIEL 4.28: Eine Methode berechnet den Mittelwert eines übergebenen *double*-Arrays.

C#

```
double Mittelwert(double[] zahlen)
{
    double sum = 0;
    int z = zahlen.Length;
    for (int i = 0; i < z; i++)
            sum += zahlen[i];
    return(sum / z);
}
```

Die Verwendung:

```
double[] zahlen = {1.5, 2, 3.8, 0.7};       // Array mit vier Werten
double mw = Mittelwert(zahlen);             // Aufruf der Methode
MessageBox.Show(mw.ToString());             // zeigt den Durchschnitt "2"
```

Rückgabe von Arrays

Methoden sind natürlich auch zur Rückgabe von Arrays in der Lage.

BEISPIEL 4.29: Die Methode liefert ein zweidimensionales *string*-Array.

C#

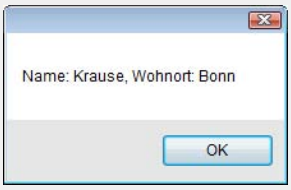
```
string[,] Kundenliste()
{
    string[,] Kunden = {{"Meier", "Berlin"}, {"Schultze", "Leipzig"},
                  {"Krause", "Bonn"}, {"Schneider", "München"}};
    return(Kunden);
}
```

BEISPIEL 4.29: Die Methode liefert ein zweidimensionales *string*-Array.

Bei der Verwendung der Methode achten Sie darauf, dass der untere Feldindex immer mit 0 beginnt:

```
string[,] liste = Kundenliste();
MessageBox.Show("Name: " + liste[2, 0] + ", Wohnort: " + liste[2, 1]);
```

Ergebnis



Ein weiteres Beispiel zur Array-Übergabe finden Sie im Praxisbeispiel

► 4.7.2 Zeichenketten mit StringBuilder addieren

4.2 Verarbeiten von Zeichenketten

Die Arbeit mit Zeichenketten (Strings) ist ein wesentlicher Bestandteil der .NET-Programmierung. Bereits im Kapitel 2 hatten Sie Möglichkeiten kennen gelernt, mit denen Konvertierungen von und in den *string*-Datentyp vorgenommen werden können. Es ist jetzt an der Zeit, dass Sie sich mit weiteren wichtigen Verfahren zur Manipulation von Zeichenketten vertraut machen.

4.2.1 Zuweisen von Strings

Ebenso wie Arrays sind auch Strings *Referenztypen*, d.h., eine String-Variable speichert nicht den Wert, sondern nur einen Verweis (Referenz, Adresse) auf die Speicherstelle. Allerdings gilt es, einen wesentlichen Unterschied festzuhalten:

HINWEIS: Mit jeder Änderung an einer *String*-Variablen wird ein neuer String erzeugt!

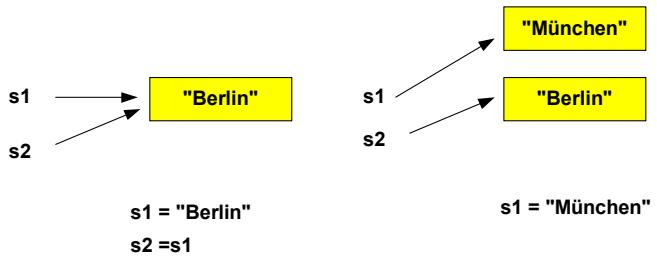
Durch diese Besonderheit wird ein Verhalten wie bei Wertetypen erreicht, obwohl es sich um Referenztypen handelt. Wer es nicht glaubt, möge bitte folgendes Beispiel ausprobieren:

BEISPIEL 4.30: String *s2* erhält einen Verweis auf String *s1*. Anschließend wird *s1* geändert.

C#

```
string s1 = "Berlin";
string s2 = s1;           // s2 referenziert s1
s1 = "München";           // legt neues String-Objekt an (ohne Auswirkung auf s2)!
MessageBox.Show(s1);       // zeigt "München"
MessageBox.Show(s2);       //  zeigt "Berlin"
```

Die folgende Abbildung soll Ihnen das Beispiel noch einmal grafisch verdeutlichen.



4.2.2 Eigenschaften und Methoden von String-Variablen

Unter .NET ist "alles ein Objekt", und so ist auch jede Zeichenkettenvariable nicht nur schlechthin vom *string*-Datentyp, sondern genaugenommen eine Instanz (ein Objekt) der Klasse *string*. Die folgende Tabelle soll einen groben Überblick über die Eigenschaften und Methoden dieser Objekte vermitteln.

Eigenschaft/Methode	Typ	Beschreibung
`Length`	*int*	... liefert die Anzahl der Zeichen
`EndsWith(string val )`	*bool*	... testet, ob String mit einem bestimmten String endet
`IndexOf(string|char val [,int startIndex[, int count])`	*int*	... sucht Position eines übergebenen Zeichens/Strings, liefert -1 wenn erfolglos
`Insert(int startIndex, string val)`	*string*	... fügt String ab *startIndex* ein
`PadLeft(int totalWidth [,char paddingChar)`	*string*	... füllt String links mit Zeichen auf, bis Gesamtbreite *totalWidth* erreicht ist
`PadRight(int totalWidth [,char paddingChar])`	*string*	... wie *PadLeft*, nur rechts
`Remove(int start, int count)`	*string*	... löscht eine Anzahl von Zeichen ab einer bestimmten Position
`Replace(char|string old, char|string)`	*string*	... ersetzt alle Zeichen bzw. Teilstrings
`Split(char separator, int count)`	*string[]*	... zerlegt String in Teilstrings auf Basis eines Trennzeichens in ein String-Array
`StartsWidth(string val)`	*bool*	... testet, ob String mit einem bestimmten String beginnt
`SubString(int start, [int len])`	*string*	... liefert einen Teilstring ab einer bestimmten Position (in einer bestimmten Länge)
`ToCharArray([int start,] int len])`	*char[]*	... liefert String (oder Teilstring) als Char-Array
`ToLower([CultureInfo cult])`	*string*	... konvertiert alle Zeichen in Kleinbuchstaben
`ToUpper([CultureInfo cult])`	*string*	... konvertiert alle Zeichen in Großbuchstaben
`Trim([char[] trimChars)`	*string*	... löscht alle vor- oder nachlaufenden Leerzeichen (oder andere)

BEISPIEL 4.31: Die Anzahl Zeichen einer *TextBox* wird in einem Meldungsfenster angezeigt.

C#
```
MessageBox.Show(textBox1.Text.Length.ToString(), "Stringlänge");
```

BEISPIEL 4.32: Die Länge eines Strings wird in einem *Label* angezeigt.

C#
```
string s = "Wie lang ist dieser Text?";
int len = s.Length;
label1.Text = len.ToString();                    // 25
```

BEISPIEL 4.33: Alles herauskopieren ab Position 10 (das erste Zeichen hat die Position 0!)

C#
```
string s = "Macht C# wirklich Spaß?";
label1.Text = s.Substring(10);                   // "irklich Spaß?"
```

BEISPIEL 4.34: Sechs Zeichen ab Position 10 herauskopieren

C#
```
string s = "Macht C# wirklich Spaß?";
label1.Text = s.Substring(10, 6);                // "irklic"
```

BEISPIEL 4.35: Suchen nach einer bestimmten Zeichenfolge

C#
```
string s = "Das Wetter ist heute warm und sonnig!";
label1.Text = s.IndexOf("warm").ToString();      // 21
```

BEISPIEL 4.36: Ersetzen einer bestimmten Zeichenfolge

C#
```
string s = "Ist .NET nicht eine Herausforderung?";
label1.Text = s.Replace("Herausforderung", "Zumutung");  // "Ist .NET nicht eine Zumutung?"
```

BEISPIEL 4.37: Mehrfaches Ersetzen

C#
```
string s = "Ist .NET nicht eine Herausforderung?";
label1.Text = s.Replace("e", "E");                    // "Ist .NET nicht EinE
HErausfordErung?"
```

BEISPIEL 4.38: In Großbuchstaben verwandeln

C#
```
string s = "Ist .NET nicht umwerfend schön?";
label1.Text = s.ToUpper();       // "IST .NET NICHT UMWERFEND SCHÖN?"
```

BEISPIEL 4.39: Das letzte Zeichen herauskopieren

C#
```
string s = "Hat sich .NET nicht toll weiterentwickelt?";
label1.Text = s.SubString(s.Length-1);       // "?"
```

Strings in Char-Arrays verwandeln

BEISPIEL 4.40: Kopieren eines Strings in ein *Char*-Array

```
C#  string s = "Hallo";
    char[] c = s.ToCharArray();
    for (int i = 0; i <= c.GetUpperBound(0); i++)
      MessageBox.Show(c[i].ToString());        // zeigt nacheinander "H", "a", "l", "l", "o"
```

BEISPIEL 4.41: Die Umkehrung des Vorgängerbeispiels, das Rückverwandeln eines *Char*-Arrays in einen *String*

```
C#  char[] c = {'H', 'a', 'l', 'l', 'o'};
    string s = "";
    for (int i=0; i < c.Length; i++)
      s = s + c[i];
    MessageBox.Show(s);                    // zeigt "Hallo"
```

4.2.3 Wichtige Methoden der String-Klasse

Die Klasse *String* verfügt über einige statische Methoden zur Bearbeitung von Zeichenketten, diese können also ohne vorhandene Instanz eines Datentyps aufgerufen werden. Einen kleinen Überblick vermittelt die folgende Tabelle.

Methode	Typ	Beschreibung
`Clone()`	*object*	... liefert Verweis auf Instanz
`Compare(string str1, string str2 [,bool ignoreCase])`	*int*	... vergleicht zwei Strings mit (oder ohne) Berücksichtigung der Groß-/Kleinschreibung
`Copy(string str)`	*string*	... erzeugt neue String-Instanz
`Empty()`	*string*	... liefert einen Leerstring ("")
`Format(string format, object arg0 [,object arg1] [,object arg2 ])`	*string*	... formatiert einen String aufgrund von Formatierungsargumenten
`Join(string separator, string[] val [, int start ,int count])`	*string*	... erzeugt aus String-Array einen einzelnen String, wobei die Teile durch Separator getrennt werden

BEISPIEL 4.42: Vergleich des Inhalts zweier TextBoxen unter Berücksichtigung der Groß-/Kleinschreibung

```
C#  int i = String.Compare(textBox1.Text, textBox2.Text, false);
    switch (i)
    {
        case -1: label1.Text = "String 1 ist kleiner als String 2 !"; break;
        case 0 : label1.Text = "String 1 ist gleich String 2 !"; break;
        case 1 : label1.Text = "String 1 ist größer als String 2 !"; break;
    }
```

Strings zerlegen und wieder zusammenstückeln

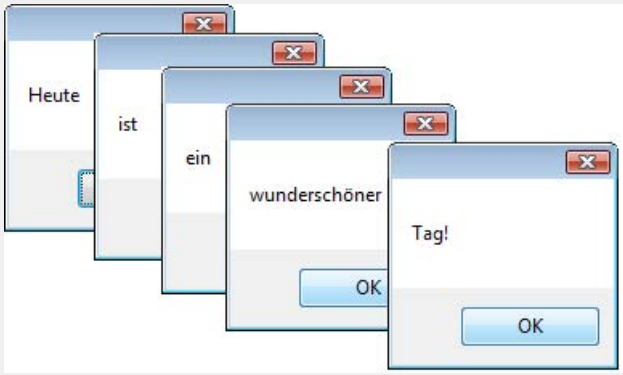
Ein gutes Beispiel für das Zusammenwirken von Instanzen- und Klassenmethoden ist das Zerlegen und Zusammenfügen von Zeichenketten. Zum Einsatz kommen die *Split*-Methode (eine Instanzenmethode) und die *Join*-Methode (eine statische Methode der *String*-Klasse).

BEISPIEL 4.43: Ein *String* wird in Teilstrings zerlegt

C#

```
string[] a;
string s = "Heute ist ein wunderschöner Tag!";
a = s.Split(' ');              // Trennzeichen ist das Leerzeichen!
for (int i = 0; i <= a.GetUpperBound(0); i++)
  MessageBox.Show(a[i]);       // zeigt Array-Elemente nacheinander an
```

Ergebnis



BEISPIEL 4.44: Der *String* wird wieder zusammengefügt, neues Trennzeichen ist "-"

C#

```
s = String.Join("-", a);
MessageBox.Show(s);
```

Ergebnis

HINWEIS: Beachten Sie, dass das Trennzeichen der *Split*-Methode vom *char*-Datentyp und das der *Join*-Methode vom *string*-Datentyp ist.

BEISPIEL 4.45: Der Inhalt einer mehrzeiligen *TextBox* (*MultiLine*-Eigenschaft ist *true*) wird zeilenweise in einem Array abgelegt und anschließend wieder zurück in die *TextBox* kopiert

C#
```
string[] a;
char c = (char) 13;                  // Trennzeichen hat Zeichencode 13 (CR)
a = textBox1.Text.Split(c);          // zeilenweise ab ins Array!
// ......
string s = c.ToString();             // Trennzeichen in string-Datentyp konvertieren
textBox2.Text = String.Join(s,a);    // rein in die zweite Textbox!
```

HINWEIS: Was es mit der *Format*-Methode auf sich hat, erfahren Sie im Abschnitt 4.6.2.

4.2.4 Die StringBuilder-Klasse

Da mit jeder Änderung an einer *String*-Variablen ein neuer String erzeugt wird, hat dies besonders dann negative Auswirkungen auf die Performance, wenn der Wert eines Strings sehr häufig geändert wird. Einen eleganten Ausweg zeigt die Klasse *StringBuilder* (Namespace *System.Text*).

Im Praxisbeispiel

► 4.7.2 Zeichenketten mit StringBuilder addieren

finden Sie einen eindrucksvollen Vergleichstest zwischen einer herkömmlichen Stringadition und der mittels eines *StringBuilder*-Objekts.

Grundlegendes

Zum Erzeugen eines *StringBuilder*-Objekts stehen Insgesamt stehen sechs Konstruktoren zur Verfügung, mit denen man in verschiedenen Kombinationen einen Initialisierungsstring und eine Anfangskapazität zuweisen kann.

BEISPIEL 4.46: Ein *StringBuilder*-Objekt für die Zeichenfolge "Heute ist Mittwoch!" wird erzeugt

C#
```
StringBuilder strB = new StringBuilder("Heute ist Mittwoch!");
```

Im obigen Konstruktor wurde die Anfangskapazität nicht explizit zugewiesen, sie wird deshalb automatisch auf einen der Standardwerte 16, 32, 64, ... festgelegt. Da die Zeichenkette 19 Zeichen lang ist wird das *StringBuilder*-Objekt hier mit der Kapazität 32 erzeugt.

Wollen Sie die im *StringBuilder*-Objekt enthaltene Zeichenfolge auslesen, so beachten Sie:

HINWEIS: Ein *StringBuilder*-Objekt repräsentiert keine Zeichenfolge (dazu dient der Datentyp *string*). Um die entsprechende Zeichenfolge zu gewinnen, müssen Sie das *StringBuilder*-Objekt (mittels *ToString()*-Methode) in einen *string* konvertieren.

BEISPIEL 4.47: Der Inhalt des im Vorgängerbeispiel erzeugten *StringBuilder*-Objekts wird ausgelesen

C#
```
MessageBox.Show(strB.ToString());
```

Ergebnis

Eigenschaften

Eigenschaft	Beschreibung
`Capacity`	... liefert oder setzt die Kapazität eines *StringBuilder*-Objekts
`Chars`	... liefert das Zeichen an einer bestimmten Position (Indexer)
`Length`	... liefert die Länge der Zeichenfolge
`MaxCapacity`	... liefert die maximale Kapazität

HINWEIS: Verwechseln Sie die Eigenschaft *Capacity* nicht mit der Ihnen bereits von der *String*-Klasse her bekannten *Length*-Eigenschaft!

BEISPIEL 4.48: Ein *StringBuilder*-Objekt wird erzeugt, dann werden diverse Eigenschaften ausgelesen

C#
```
StringBuilder strB = new StringBuilder("Heute ist Mittwoch!", 50);
int cap = strB.Capacity;          //  50
int lg = strB.Length;             //  19
int max = strB.MaxCapacity;       //  2147483647
```

BEISPIEL 4.49: Aus der Zeichenfolge "Heute ist Mittwoch" wird das Zeichen an der 11. Position geliefert

C#
```
StringBuilder strB = new StringBuilder("Heute ist Mittwoch");
char c = strB[10];
label1.Text = c.ToString();      // "M"
```

Methoden

Methode	Beschreibung
`Append`	... fügt eine Zeichenkette zu einem *StringBuilder*-Objekt hinzu
`AppendFormat`	... fügt eine formatierte Zeichenkette hinzu
`AppendLine`	... fügt eine Zeile hinzu
`CopyTo`	... kopiert einen Teil des *StringBuilder*-Objekts in ein *Char*-Array
`EnsureCapacity`	... sichert ab, dass das *StringBuilder*-Objekts so groß ist wie angegeben
`Insert`	... fügt ab einer bestimmten Position eine Zeichenkette ein
`Remove`	... löscht aus der Zeichenfolge eine Zeichenfolge ab einer bestimmten Position
`Replace`	... ersetzt in der Zeichenfolge ein Zeichen durch ein anderes

BEISPIEL 4.50: Eine Zeichenkette wird einem *StringBuilder*-Objekt zugewiesen

C#
```
StringBuilder strB = new StringBuilder();
strB = strB.Append("Heute ist Mittwoch!");
label1.Text = strB.ToString();               // "Heute ist Mittwoch!"
```

BEISPIEL 4.51: In die Zeichenfolge "Heute !" wird ab Position 6 die Zeichenfolge "ist Mittwoch" eingefügt

C#
```
StringBuilder strB = new StringBuilder("Heute !");
strB = strB.Insert(6, "ist Mittwoch");
label1.Text = strB.ToString();                // "Heute ist Mittwoch!"
```

BEISPIEL 4.52: In der Zeichenfolge "Heute ist Mittwoch!" wird das letzte Zeichen gelöscht

C#
```
StringBuilder strB = new StringBuilder("Heute ist Mittwoch!");
strB.Remove(strB.Length-1,1);
label1.Text = strB.ToString();                // "Heute ist Mittwoch"
```

BEISPIEL 4.53: In der Zeichenfolge "Heute ist Mittwoch!" wird der Wochentag ersetzt

C#
```
StringBuilder strB = new StringBuilder("Heute ist Mittwoch!");
strB.Replace("Mittwoch", "Donnerstag");
label1.Text = strB.ToString();                // "Heute ist Donnerstag!"
```

4.3 Reguläre Ausdrücke

Sie haben im vorhergehenden Teilkapitel zahlreiche (mehr oder weniger raffinierte) Methoden der Zeichenkettenverarbeitung kennengelernt. Einfache Aufgaben, wie zum Beispiel das Ersetzen eines Strings durch einen anderen oder die Suche nach einem bestimmten Teilstring, lassen sich damit relativ einfach lösen (z.B. *Replace, IndexOf ...*). Bei anspruchsvolleren Problemen, wie zum Beispiel die Gültigkeitsüberprüfung einer EMail-Adresse, sind Sie jedoch ziemlich schnell mit Ihrem

Latein am Ende. Früher oder später werden Sie deshalb auf die Klassen für reguläre Ausdrücke aus dem Namespace *System.Text.RegularExpressions* stoßen.

Ein *regulärer Ausdruck* bietet eine flexible und effiziente Methode zur Verarbeitung von Text. Die Möglichkeiten sind so umfassend (es handelt sich gewissermaßen um eine eigene Sprache!), dass wir hier lediglich Ihr Interesse für dieses mächtige Werkzeug wecken und die unserer Meinung nach für den Einstieg wichtigsten Sprachelemente vorstellen können.

HINWEIS: Für das vertiefende Selbststudium gibt es genügend Quellen (Spezialliteratur, .NET-Framework- Dokumentation).

4.3.1 Wozu werden reguläre Ausdrücke verwendet?

Häufig werden reguläre Ausdrücke als Filterkriterien in der Textsuche benutzt, wobei der Text mit dem Muster des regulären Ausdrucks abgeglichen wird (so genanntes Pattern Matching). Wichtige Einsatzgebiete sind:

- Überprüfung von Texteingaben (z.B. Passwörter und Telefonnummern)
- Texte in strukturiertere Formate zerlegen (z.B. Daten von einer HTML Seite extrahieren und in eine Datenbank ablegen)
- Bestimmte Textmuster in einem Dokument ersetzen (z.B. nur ganze Wörter, die mit K beginnen und mit E enden)

Aber nicht nur analytische, auch synthetische Aufgaben können mittels regulärer Ausdrücke gelöst werden. Man kann damit automatisch Mengen von Wörtern erzeugen (z.B. bei einer gegebenen maximalen Zeichenanzahl alle denkbaren Wörter generieren, die mit K beginnen und mit E enden)[1].

4.3.2 Eine kleine Einführung

Womit anfangen? Beginnen wir einfach mal mit einem der gebräuchlichsten Operatoren in regulären Ausdrücken, dem ?-Quantifizierer, der ein optionales Vorkommen des Vorgängerelements definiert (das Element ist genau einmal oder überhaupt nicht vorhanden).

Wenn Sie sich nun die folgende Tabelle betrachten, so dürfte Ihnen das allgemeine Prinzip regulärer Ausdrücke einleuchten:

Regulärer Ausdruck	Beispiel für Teststring	Ergebnis
`Walth?er`	Walter	Treffer
	Walther	Treffer
	Waalter	kein Treffer

[1] SPAM-Schleudern generieren so systematisch E-Mail-Adressen (den Teil vor dem @).

4.3.3 Wichtige Methoden/Eigenschaften der Klasse Regex

Der Namespace *System.Text.RegularExpressions* besteht aus mehreren Klassen, die wichtigste ist *Regex*. Diese repräsentiert einen unveränderlichen regulären Ausdruck und dient dem Erzeugen und Überprüfen regulärer Ausdrücke.

Die folgende Tabelle fasst die wichtigsten Methoden zusammen:

Methode	Beschreibung
IsMatch(str)	... liefert *true/false*
Match(str)	... liefert einen Treffer
Matches(str)	... liefert eine *MatchCollection*
Replace(str1, str2)	... ersetzt Vorkommen von *str1* durch *str2*
Split(str)	... trennt mit *str* zwischen den Fundstellen des regulären Ausdrucks

Die Match-Methode

Für die Überprüfung regulärer Ausdrücke können wir die statische *Match*-Methode der *RegEx*-Klasse verwenden:

BEISPIEL 4.54: Der reguläre Ausdruck "Walth?er" wird getestet

C#
```
...
using System.Text.RegularExpressions;
...
string rgEx = @"Walth?er";
string testStr = "Waalther";            // mit verschiedenen Strings testen
if (Regex.Match(testStr, rgEx).Success)
    MessageBox.Show("Treffer!");
else
    MessageBox.Show("Kein Treffer!");
```

HINWEIS: Durch das Voranstellen von @ vor den regulären Ausdruck wird die Escape-Funktion von C# unterbunden, wir gehen an späterer Stelle genauer darauf ein[1].

[1] Obwohl das @ nicht bei allen regulären Ausdrücken erforderlich wäre, gewöhnen wir uns lieber gleich daran.

Eigenschaften des Match-Objekts

Das von der *Match*-Methode gelieferte *Match*-Objekt kapselt Eigenschaften für den aktuellen Wert, dessen Index und Länge.

BEISPIEL 4.55: Eigenschaften des Match-Objekts ermitteln

C#
```
Match m = Regex.Match ("Walter geht in den Garten.", @"Walth?er");
m.Success;          // True
m.Value;            // Walter
m.Index;            // 0
m.Length;           // 6
```

Im Vergleich mit der Ihnen aus der klassischen Zeichenkettenverarbeitung bekannten *IndexOf*-Methode ist die *Match*-Methode die leistungsfähigere Variante. Der Unterschied ist, dass jetzt nach einem Muster gesucht wird statt nach einer Zeichenkette.

Die NextMatch-Methode

Standardmäßig wird von links nach rechts gesucht, deshalb wird nur der äußerste linke Treffer zurückgegeben. Verwenden Sie deshalb die *NextMatch*-Methode um mehr Treffer zu erhalten.

BEISPIEL 4.56: NextMatch

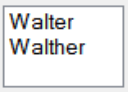
C#
```
Match m1 = Regex.Match("Walter oder Walther", @"Walth?er");
Match m2 = m1.NextMatch();
label1.Text = m1.ToString();          // Walter
label2.Text = m2.ToString();          // Walther
```

Die Matches-Methode

Die *Matches*-Methode liefert alle Treffer als Array zurück.

BEISPIEL 4.57: Eine erweiterte Version des Vorgängerbeispiels

C#
```
foreach (Match m in Regex.Matches("Walter oder Walther", @"Walth?er"))
    listBox1.Items.Add(m.ToString());
```



Der Alternator

Ein anderer sehr gebräuchlicher Operator für reguläre Ausdrücke ist der Alternator, ausgedrückt durch einen vertikalen Strich (|). Die Klammern um einen Alternator trennen die Alternativen vom Rest des Ausdrucks.

BEISPIEL 4.58: Der reguläre Ausdruck "Wal(t|th)?er" wird getestet

C#
```
Regex.IsMatch("Walther", @"Wal(t|th)?er");      // True
Regex.IsMatch("Walter",  @"Wal(t|th)?er");      // True
Regex.IsMatch("Waler", @"Wal(t|th)?er");        // True
Regex.IsMatch("Waltter", @"Wal(t|th)?er");      // False
Regex.IsMatch("Walher", @"Wal(t|th)?er");       // False
Regex.IsMatch("Wal", @"Wal(t|th)?er");          // False
```

Die *IsMatch*-Methode (siehe obiges Beispiel) stellt gewissermaßen eine Zusammenfassung der *Match*-Methode und der *Success*-Eigenschaft dar.

4.3.4 Kompilierte reguläre Ausdrücke

In einigen der Vorgängerbeispiele haben wir statische *Regex*-Methoden wiederholt mit dem gleichen regulären Ausdruck aufgerufen. Genauso könnten wir aber auch ein *Regex*-Objekt mit diesem Pattern instanziieren und dann die Instanzenmethoden aufrufen.

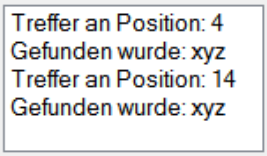
BEISPIEL 4.59: Eine kompilierte Variante des Vorgängerbeispiels

C#
```
Regex r = new Regex(@"Wal(t|th)?er");
r.IsMatch("Walter");         // True
r.IsMatch("Waltter");        // False
```

Beiden Varianten (Klassen-/Instanzenmethode) unterscheiden sich hinsichtlich der Performance. Eine *Regex*-Instanz verwendet im Hintergrund eine leichtgewichtige Code-Generierung um den erforderlichen Code dynamisch zu erzeugen und zu kompilieren. Dies kann bis zu 10mal schneller sein.

BEISPIEL 4.60: Alle Vorkommen von "xyz" in einer Zeichenkette werden ermittelt und angezeigt

C#
```
Regex rx = new Regex("xyz");
MatchCollection mcoll = rx.Matches("0123xyz4567uvwxyza");
for (int i = 0; i < mcoll.Count; i++)
{
    listBox1.Items.Add("Treffer an Position: " + mcoll[i].Index);
    listBox1.Items.Add("Gefunden wurde: " + mcoll[i].Value);
}
```



HINWEIS: Eine *Regex*-Instanz ist unveränderlich!

4.3.5 RegexOptions-Enumeration

Mit den Flags von *RegexOptions* können Sie das Suchverhalten beeinflussen. Die folgende Tabelle zeigt einige wichtige Mitglieder dieser Enumeration:

Name	Code	Beschreibung
IgnoreCase	i	... ignoriert die Groß-Kleinschreibung (standardmäßig ausgeschaltet)
Multiline	m	... ändert ^ und $, sodass diese sich auf Anfang und Ende einer Zeile statt eines Strings beziehen
ExplicitCapture	n	... erfasst nur explizit benannte Gruppen
IgnorePatternWhitespace	x	... eliminiert Leerzeichen aus dem Muster
RightToLeft	r	... sucht von rechts nach links
CultureInvariant		... kein kulturspezifisches Verhalten beim Stringvergleich
Compiled	c	... erzielt Performance-Gewinn (verlangsamt aber den Programmstart)

BEISPIEL 4.61: Suche ohne Berücksichtigung der Groß-/Kleinschreibung

C#
```
Match m = Regex.Match ("walter", @"Walth?er", RegexOptions.IgnoreCase);
label1.Text = m.ToString();          //  "walter"
```

Die meisten der *RegexOptions* können innerhalb eines regulären Ausdrucks mittels eines einzelnen Zeichencodes aktiviert werden, damit lässt sich etwas Schreibarbeit sparen.

BEISPIEL 4.62: Verkürzte Notation des Vorgängerbeispiels mittels des Zeichencodes (?i)

C#
```
Match m = Regex.Match ("walter", @"(?i)Walth?er");
label1.Text = m.ToString();              //  "walter"
```

Oft benutzt man auch die Option*IgnorePatternWhitespace* oder (?x). Damit wird Ihnen das Einfügen von Leerzeichen ermöglicht, um einen regulären Ausdruck lesbarer zu machen (die Leerzeichen werden dabei nicht als Literale interpretiert).

BEISPIEL 4.63: Anwendung der Option *IgnorePatternWhitespace* (?x)

C#
```
Regex.Match ("Walter", @"(?x)W a l t h ? e r");            // liefert "Walter"
```

4.3.6 Metazeichen (Escape-Zeichen)

Die folgenden Zeichen haben innerhalb von regulären Ausdrücken eine spezielle Bedeutung und man tut gut daran, sie auswendig zu lernen:

\ * + ? | { [( ) ^ $. #

Diese so genannten Metazeichen verändern die Bedeutung von normalen Zeichen. Das Pluszeichen (+) ist beispielsweise ein so genannter Quantifizierer (Existenzquantor, siehe dazu den übernächsten Abschnitt).

HINWEIS: Um einem Metazeichen wieder seine ursprüngliche Bedeutung als Zeichenliteral zurückzugeben, muss ein Backslash (\) vorangestellt werden.

BEISPIEL 4.64: Der reguläre Ausdruck "1+1" richtig und falsch geschrieben

C#
```
Regex.Match ("1+1=2", @"1\+1");       // "1+1"    (richtig)
Regex.Match ("1+1=2", @"1+1"));       // ""       (falsch)
```

HINWEIS: Falls sich ein Metazeichen innerhalb einer in eckige Klammern [ ] eingeschlossenen Zeichenmenge befindet gilt obige Regel nicht (siehe folgender Abschnitt).

Die Methoden *Escape* und *Unescape* konvertieren einen String, der Metazeichen enthält, in einen regulären Ausdruck, in welchem die Metazeichen mittels Backslash "escaped" werden (und umgekehrt).

BEISPIEL 4.65: Die Methoden *Escape* und *Unescape*

C#
```
...
label1.Text = Regex.Escape("1+1=2");              // "1\+1=2"
label2.Text = Regex.Unescape (@"1\+1=2");         // "1+1=2"
```

Was wir bisher bereits getan haben, soll noch einmal besonders unterstrichen werden:

HINWEIS: In unseren Beispielen werden wir allen regulären Ausdrücken das @ -Zeichen voranstellen[1]!

4.3.7 Zeichenmengen (Character Sets)

Charactersets stellen Pattern (Wildcards) für bestimmte zu filternde Zeichenfolgen dar. Die folgende Tabelle verdeutlicht die Syntax:

Ausdruck	Beschreibung	Negation
[abcdef]	... trifft ein einzelnes Zeichen in der Liste	[^abcdef]
[a-f]	... trifft ein einzelnes Zeichen in einem Bereich	
\d	... trifft ein Dezimalzeichen (entspricht [0-9])	\D
\w	... trifft ein Zeichen in einem Wort (abhängig von *CultureInfo.CurrentCulture*; z.B. dasselbe wie [a-zA-Z_0-9])	\W

[1] Dadurch deaktivieren wir den Escape-Mechanismus von C#, der ebenfalls den Backslash nutzt.

Ausdruck	Beschreibung	Negation
\s	... trifft ein Leerzeichen (dasselbe wie [\n\r\t\f])	\S
.	... trifft standardmäßig jedes Zeichen außer \n	\n
\b	... in einem regulären Ausdruck eine Wortgrenze, in einer Zeichenmenge a[ ] das Backspace-Zeichen!	

Steht beispielsweise ein ^ am Anfang einer Zeichenmenge, so wird diese Menge negiert.

BEISPIEL 4.66: Der gesuchte String muss 2 Zeichen lang sein und mit "W" beginnen, danach kann ein beliebiges Zeichen außer "a" und "e" folgen

C#
```
Match m = Regex.Match("Walter,Werner,Wolfgang", @"W[^ae]");
int i = m.Index.ToString();       // 14
string s = m.Value;               // "Wo"
```

Um ein einzelnes Zeichen aus einer Zeichenmenge zu treffen, muss die Zeichenmenge in eckige Klammern eingeschlossen werden:

BEISPIEL 4.67: Wieviel Strings "Walter" oder "walter" sind enthalten?

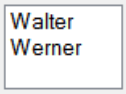
C#
```
int i = Regex.Matches ("Walter,wilhelm,walter.", @"[Ww]alter").Count;   // 2
```

HINWEIS: Der Ausdruck *[A-Za-z0-9]* ist die Kurzschreibweise für *[A-Z][a-z][0-9]*.

In regulären Ausdrücken gehört der Punkt (.) zu den am häufigsten benutzten Metazeichen. Er steht für ein beliebiges einzelnes Zeichen mit Ausnahme des Zeilenumbruchs (Newline bzw. *\n*). Logischerweise trifft dann der Punkt auf die negierte Zeichenklasse *[^\r\n]* zu.

BEISPIEL 4.68: Vornamen, die mit "W" beginnen, mit "er" enden und in der Mitte drei beliebige Zeichen haben

C#
```
foreach (Match m in Regex.Matches("Walter,Werner,Wolfgang", @"W...er"))
   listBox1.Items.Add(m.ToString());
```



Einen Zeichenbereich gibt man mittels Bindestrich an.

BEISPIEL 4.69: Die Gültigkeit eines Schachzugs überprüfen[1]

C#
```
if (Regex.Match ("c2-c4", @"[a-h]\d-[a-h]\d").Success)
        MessageBox.Show("Der Zug ist gültig!");
```

[1] Dieses einfache Pattern entspricht nur bedingt den Schachregeln, aber uns geht es ja hier nur um das Prinzip.

4.3.8 Quantifizierer

Quantifizierer beziehen sich auf den direkt davor stehenden Ausdruck und bestimmen die minimale, maximale oder exakte Anzahl von Wiederholungen. In unseren Einführungsbeispielen hatten wir bereits ausgiebigen Gebrauch vom Quantifizierer ? gemacht, die folgende Tabelle gibt eine Übersicht über alle wichtigen Quantifizierer.

Quantifier	Beschreibung	Beispiel
*	Zeichen kommt beliebig oft vor (Allquantor)	*a**
+	mindestens einmaliges Vorkommen (Existenzquantor)	*a+*
?	genau einmal oder kein Vorkommen	*ab?a*
{n}	genau n- maliges Vorkommen	*a{4}*
{n,}	mindestens n- maliges Vorkommen	*a{3,}*
{,m}	maximal n- maliges Vorkommen	*a{,2}*
{n,m}	mindestens n- maliges, maximal m- maliges Vorkommen	*a{1,5}*

Der Quantifizierer * legt fest, dass das vorhergehende Zeichen (oder die Gruppe) überhaupt nicht oder aber beliebig oft vorkommen darf.

BEISPIEL 4.70: Es wird überprüft, ob der Dateiname *bild5.jpg* zu der Serie *bild0.jpg, bild1.jpg, ...* gehört

C#
```
bool suc = Regex.Match ("bild5.jpg", @"bild\d*\.jpg").Succes;              // True
```

HINWEIS: Beachten Sie den Backslash \ vor dem Punkt in der Datei-Extension, er verhindert die Interpretation des Punkts als Metazeichen!

BEISPIEL 4.71: Es werden alle mit *bild...* beginnenden *.jpg*-Dateien erfasst

C#
```
bool suc = Regex.Match ("bildUrlaub_12.jpg", @"bild.*\.jpg").Success;       // True
```

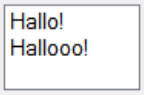
HINWEIS: Beachten Sie, dass hier ein beliebig oft vorkommendes Zeichen mit .* (und nicht *) definiert ist (der Punkt als Metazeichen mit nachfolgendem Quantifizierer).

Der Quantifizierer + erlaubt das ein- oder mehrmalige Vorkommen des vorhergehenden Zeichens (oder der Gruppe).

BEISPIEL 4.72: Alle "Hallo...!" mit einem oder mehreren "o" werden herausgefiltert

C#
```
foreach (Match m in Regex.Matches("Hallo! Hallo Hallooo!", @"Hallo+!"))
            listBox1.Items.Add(m.ToString());
```

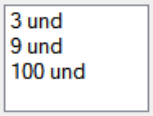
BEISPIEL 4.72: Alle "Hallo...!" mit einem oder mehreren "o" werden herausgefiltert



Der {} Quantifizierer legt eine Anzahl oder einen Bereich für die Wiederholungen fest.

BEISPIEL 4.73: Es werden die Strings herausgefiltert, die aus einer ein- oder dreistelligen Zahl bestehen, gefolgt von " und "

C#
```
foreach (Match m in Regex.Matches("3 und 9 und 100 und 12 ergibt 124", (@"\d{1,3} und")))
    listBox1.Items.Add(m.ToString());
```



4.3.9 Zero-Width Assertions

Zu den Zero-Width Assertions[1] gehören Elemente wie Ankerzeichen für String-, Zeilen- und Wortgrenzen sowie Lookahead/Lookbehind-Ausdrücke. Der etwas merkwürdige Begriff soll aussagen, dass die Breite eines Treffers durch das Einfügen dieser Elemente nicht vergrößert wird.

String- und Zeilengrenzen (^ und $)

Bislang hatten wir es in regulären Ausdrücken nur mit Zeichenliteralen und Zeichenmengen zu tun, und die Regex-Engine suchte nach Übereinstimmungen mit einzelnen Zeichen. Ankerzeichen hingegen stimmen mit keinem Zeichen überein, sondern stattdessen mit einer Position vor, nach oder zwischen einzelnen Zeichen, und sie können zum "verankern" des Treffers an einer bestimmten Position benutzt werden. Das Ankerzeichen ^ hat die Position vor dem ersten Zeichen des Strings[2]. Analog dazu trifft das Ankerzeichen $ die Position nach dem letzten Zeichen.

Regulärer Ausdruck	Teststring	Treffer
`^a`	abc	a
`^b`		*kein Match!*
`c$`		c
`a$`		*kein Match!*
`^[Hh]a`	Hallo hallo!	Ha
`[Ff]$`	f = 0.5F	F

[1] "Behauptungen der Breite null"

[2] Eine andere Bedeutung von ^ ist die eines Negationsoperators für Zeichenmengen (siehe Abschnitt 4.3.7).

Wenn Sie *RegexOptions.Multiline* wählen (oder *(?m)* in den Ausdruck einfügen) bezieht sich ^ auf den Start des Strings oder der Zeile direkt nach einem Zeilenumbruch, hingegen bezieht sich (\n). *$* auf das Ende des Strings oder der Zeile direkt vor einem \n.

BEISPIEL 4.74: Alle leeren Zeilen in einem mehrzeiligen Text werden erfasst

C#
```
MatchCollection emptLines = Regex.Matches (txt, "^(?=\r?$)", RegexOptions.Multiline);
```
oder
```
MatchCollection emptLines = Regex.Matches (txt, "(?m)^(?=\r?$)");
```

Wortgrenzen (\b)

Genau wie ^ und $ ist auch \b ein Anker, allerdings nicht für Strings, sondern für Wortgrenzen. Es gibt drei unterschiedliche Positionen, die Wortgrenzen sein können:

- Vor dem ersten Zeichen im String, wenn das erste Zeichen ein Wortzeichen ist (\w).
- Nach dem letzten Zeichen im String, wenn das letzte Zeichen ein Wortzeichen ist.
- Zwischen zwei Zeichen im String, wenn das eine ein Wortzeichen ist und das andere nicht.

Der Wortanker \b wird häufig benutzt um ganze Wörter zu herauszufiltern.

BEISPIEL 4.75: Alle Wörter werden aus dem String extrahiert

C#
```
foreach (Match m in Regex.Matches("D-04610 Wintersdorf", @"\b\w+\b"))
      listBox1.Items.Add(m.Value);
```

Das Pattern \w+ innerhalb der Wortbegrenzer \b ist die Kombination einer Zeichenmenge (\w) mit dem Existenzquantor + und bedeutet die mindestens einmalige Wiederholung eines Wortzeichens.

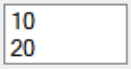
Positiver Lookahead (?=expr)

Das Konstrukt (?=expr) überprüft, ob der Text nach dem Treffer mit dem Ausdruck *expr* übereinstimmt (*expr* erscheint dabei nicht im Resultat!). Im folgenden Beispiel arbeiten wir zur Abwechslung mal wieder mit einer *Regex*-Instanz:

BEISPIEL 4.76: Das Pattern "\s(?=EUR)" steht für ein Leerzeichen, dem der String "EUR" folgt.

C#
```
Regex rx = new Regex(@"\d+\s(?=EUR)");
MatchCollection mcoll = rx.Matches("Heute 10 EUR, morgen 15EUR, übermorgen 20 EUR");
for (int i = 0; i < mcoll.Count; i++)
        listBox1.Items.Add(mcoll[i].Value);
```

BEISPIEL 4.76: Das Pattern "\s(?=EUR)" steht für ein Leerzeichen, dem der String "EUR" folgt.



Obwohl als Suchkriterium angegeben, gehört der String "EUR" im obigen Beispiel nicht zur gematchten Zeichenkette!

Das folgende Beispiel hat durchaus praktische Relevanz:

BEISPIEL 4.77: Ein Passwort soll mindestens 8 Zeichen enthalten, davon mindestens eine Ziffer

C#
```
string password = "geheim457";
bool ok = Regex.IsMatch(password, @"(?=.*\d).{8,}");      // True
```

Zunächst wurde eine Vorschau durchgeführt, ob irgendwo im String eine Ziffer auftaucht, vor der kein oder beliebig viele Zeichen stehen. Wenn ja, dann wird nach einem Treffer mit acht oder mehr Zeichen gesucht.

Vielleicht kommen auch Sie auf die Idee, im obigen Lookahead-Ausdruck .*\d den Quantifizierer * durch + auszutauschen, aber dann dürfte ein Passwort nicht mit einer Ziffer beginnen[1].

Negativer Lookahead (?!expr)

Hier wird verlangt, dass der Ausdruck nach dem Treffer **nicht** mit *expr* übereinstimmt.

Das folgende und nicht ganz ernst zu nehmende Beispiel versetzt Sie in die Rolle des verzweifelten Personalchefs einer Firma, dem Beurteilungen von unzähligen Bewerbern vorliegen.

BEISPIEL 4.78: Suche nach dem Wort "Aufgaben", wenn danach nicht "fleißig" oder "Fleiß" vorkommen

C#
```
string rex = @"Aufgaben(?!.*(fleißig|Fleiß))";
string text = "Die übertragenen Aufgaben erfüllte er mit viel Fleiß.";
Match m = Regex.Match (text, rex);
if (m.Success)
      MessageBox.Show("Bewerber ist geeignet!", "Treffer", MessageBoxButtons.OK,
                                                          MessageBoxIcon.Exclamation);
else
     MessageBox.Show("Bewerber ist ungeeignet!", "Kein Treffer", MessageBoxButtons.OK,
                                                            MessageBoxIcon.Error);
```

[1] Finden Sie das Warum selbst heraus!

BEISPIEL 4.78: Suche nach dem Wort "Aufgaben", wenn danach nicht "fleißig" oder "Fleiß" vorkommen

Das Wort "Aufgaben" wird nur dann gefunden, wenn danach nicht irgendwann im Text das Wort "fleißig" oder "Fleiß" vorkommt.

Positiver (?<=expr) und negativer (?<!expr) Lookbehind

Das Konstrukt (?<=expr) bedeutet, dass vor einem Treffer ein spezieller Ausdruck erfüllt sein muss, die Negation gilt für das Konstrukt (?<!expr).

BEISPIEL 4.79: Es wird nach "gut" gesucht, zuvor darf aber nicht "noch..." stehen.

C#
```
string rex = "(?i)(?<!noch.*)gut";
string text = "Die Leistung wird noch als Gut eingeschätzt...";

Match m = Regex.Match (text, rex);
bool b = m.Success;                          // False

text = "Die Leistung ist sehr gut, vielen Dank!";
m = Regex.Match (text, rex);
b = m.Success;                               // True
```

Das dem regulären Ausdruck vorangestellte (?i) bewirkt, dass die Groß-/Kleinschreibung bei der Suche nach "gut" ignoriert wird. Einen Treffer gibt es nur dann, wenn irgendwo vor dem "gut" nicht der Text "noch" auftaucht. Nach dem "noch" dürfen noch beliebig viele Zeichen stehen (.*).

HINWEIS: Eine weitere Verbesserung dieses Beispiels, wie auch der vorangegangenen, kann durch das Hinzufügen von Wortgrenzen (siehe Seite 268) erreicht werden.

4.3.10 Gruppen

Mitunter wollen Sie einen regulären Ausdruck in Unterausdrücke bzw. Gruppen aufteilen. Das lässt sich mittels runder Klammern erreichen, in die jede einzelne Gruppe eingeschlossen wird.

Die Gruppe mit dem Index 0 repräsentiert den gesamten Treffer, hat also denselben Wert wie der von *Match*.

BEISPIEL 4.80: Wir wollen eine deutsche Postleitzahl in zwei Gruppen aufteilen: PLZ und Ortsnamen

```
C#
Match m = Regex.Match("04610 Wintersdorf", @"(\d{5}) (\w.+)");
label1.Text = m.Groups[1].Value;       // 04610
label2.Text = m.Groups[2].Value;       // Wintersdorf
label3.Text = m.Groups[0].Value;       // 04610 Wintersdorf
```

Sie können sich sogar innerhalb eines regulären Ausdrucks auf eine Gruppe beziehen, indem Sie einen Backslash \ gefolgt von der Gruppennummer eingeben (\1 im folgenden Beispiel).

BEISPIEL 4.81: Suche nach allen Wörtern, die mit denselben Buchstaben beginnen und enden. Der Ausdruck \1 bezieht sich auf das aktuelle Wortzeichen der ersten Gruppe (\w).

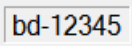
```
C#
foreach (Match m in Regex.Matches ("otto Otto ende NAMEN", @"\b(\w)\w+\1\b"))
   listBox1.Items.Add(m.Value);
```

4.3.11 Text ersetzen

Die statische Methode *RegEx.Replace* ähnelt der Ihnen bereits bekannten Instanzenmethode *Replace* einer herkömmlichen Stringvariablen, allerdings verwenden wir jetzt als Suchkriterium einen regulären Ausdruck.

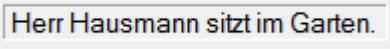
BEISPIEL 4.82: Die Großbuchstaben eines Strings werden entfernt

```
C#
string input = "AbCdE-12345";
string find = @"[A-Z]";
string replace = "";
label1.Text = Regex.Replace(input, find, replace);
```



BEISPIEL 4.83: Ersetzen von "Haus" durch "Garten"

```
C#
label1.Text = Regex.Replace("Herr Hausmann sitzt im Haus.", @"\bHaus\b", "Garten");
```



Weil wir im obigen Beispiel die Suche auf ganze Wörter begrenzen (Wortanker \b), erhalten wir nicht das Ergebnis "*Herr Gartenmann sitzt im Garten.*".

Der Ersetzungsstring kann sich mittels der Substitution $0 auf den Original-Treffer beziehen.

BEISPIEL 4.84: Alle Zahlen innerhalb eines Strings werden in spitze Klammern eingeschlossen.

C#
```
label1.Text = Regex.Replace("7 mal 8 ergibt 56", @"\d+", @"<$0>");
```

<7> mal <8> ergibt <56>

4.3.12 Text splitten

Im Vergleich mit der Ihnen bereits aus der herkömmlichen Stringverarbeitung bekannten *Split*-Methode ist die statische Methode *Regex.Split* die leistungsfähigere Variante. *Regex.Split* verwendet zur Definition des Trennmusters einen regulären Ausdruck.

BEISPIEL 4.85: Splitten einer Zeichenkette. Trennzeichen ist irgendeine Ziffer.

C#
```
foreach (string s in Regex.Split ("abc36b5cd", @"\d"))
         listBox1.Items.Add(s);
```

Ergebnis

abc

b
cd

Die Leerzeile entsteht dadurch, dass vor dem Trennzeichen "6" kein Textzeichen steht.

Trennzeichen im Ergebnis

Damit das Resultat eines Textsplittings auch die Trennelemente enthält, können wir den Ausdruck in einen positiven Lookahead (siehe Seite 268) einschließen.

BEISPIEL 4.86: Trennelemente sind die Großbuchstaben. Diese aber sollen im Ergebnis mit erscheinen.

C#
```
string rex = @"(?=[A-Z])";
string text = "JanuarFebruar März";
foreach (string s in Regex.Split(text, rex))
       listBox1.Items.Add(s);
```

Ergebnis

Januar
Februar
März

Weitere Erfahrungen im Umgang mit regulären Ausdrücken sammeln Sie im Praxisbeispiel

► 4.7.3 Reguläre Ausdrücke testen

4.4 Datums- und Zeitberechnungen

In diesem Abschnitt wollen wir uns den zahlreichen Eigenschaften und Methoden der *DateTime*-Struktur zuwenden, auf deren Grundlage Datums-/Zeitberechnungen möglich sind.

4.4.1 Die DateTime-Struktur

Dieser Datentyp wird benutzt, um ein Datum einschließlich einer Uhrzeit in einer Variablen zu speichern.

Grundlegendes

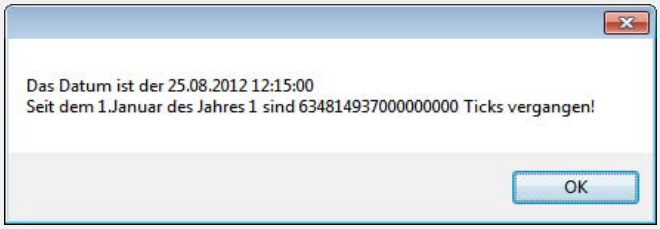
Eine Variable des Typs *DateTime* kann einen Zeitpunkt zwischen dem 1.Januar des Jahres 1 und dem 31.Dezember des Jahres 9999 darstellen[1]. Der *Tick* ist die dabei verwendete Einheit zur Messung der Zeitwerte. Die Länge eines Ticks beträgt winzige 100 Nanosekunden (0,0000001 sek!), was äußerst präzise Angaben von Zeitpunkten (bzw. Zeitmessungen) ermöglicht.

Zum Erzeugen einer *DateTime*-Variablen stehen insgesamt zwölf Konstruktoren zur Verfügung, mit denen man in verschiedensten Kombinationen ein konkretes Datum (mit und ohne Uhrzeit) zuweisen kann[2].

BEISPIEL 4.87: Eine *DateTime*-Variable für den *25.August 2012 12:15* erzeugen und auswerten

C#
```
DateTime dat1 = new DateTime(2012, 8, 25, 12, 15, 0);
long ts = dat1.Ticks;
string s = "Das Datum ist der " + dat1.ToString();
s += "\nSeit dem 1.Januar des Jahres 1 sind " + ts.ToString() + " Ticks vergangen!";
MessageBox.Show(s);
```

Ergebnis



Um aus den ermittelten Datum und Uhrzeit zu rekonstruieren, verwenden wir einfach einen anderen Konstruktor:

BEISPIEL 4.88: Fortsetzung des Vorgängerbeispiels

C#
```
DateTime dat2 = new DateTime(ts);
MessageBox.Show(dat2.ToString());           // "25.08.2012 12:15:00"
```

[1] Gregorianischer Kalender

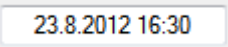
[2] Sie können auch festlegen, ob sich die Zeitangabe auf die lokale Systemzeit, die koordinierte Weltzeit (UTC) oder eine andere beziehen soll.

Nachdem Sie nun einen ersten Eindruck vom Umgang mit *DateTime* gewonnen haben, wollen wir uns im Folgenden den zahlreichen Eigenschaften und Methoden genauer zuwenden. Dabei müssen wir -- wie bereits bei der *Array*- und der *String*-Klasse gezeigt -- unterscheiden, ob die Eigenschaften/Methoden auf eine *DateTime*-Variable oder auf die *DateTime*-Struktur angewendet werden.

4.4.2 Wichtige Eigenschaften von DateTime-Variablen

Eigenschaft	Beschreibung
`Ticks`	... liefert die Anzahl von *Ticks* einer *DateTime*-Variablen(*long*-Typ)
`Second, Minute, Hour, Day, Month, Year`	... liefern die Bestandteile eines Datums als *Integer*-Zahl.
`DayOfWeek`	... liefert den Wochentag als *DayOfWeek*-Enumeration (*Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday*).
`DayOfYear`	... liefert die Nummer des Tages im Jahr als *Integer*-Zahl.

BEISPIEL 4.89: Die Eigenschaften eines Datums, welches in eine *TextBox* eingegeben wurde



C#
```
DateTime dat = Convert.ToDateTime(textBox1.Text);
int h = dat.Hour;               // h erhält den Wert 16
int m = dat.Minute;             // m erhält den Wert 30
int y = dat.Year;               // y erhält den Wert 2012
```

BEISPIEL 4.90: Der Wochentag für das Datum 23.8.2012 wird ermittelt

C#
```
DateTime dat = Convert.ToDateTime("23.8.2012");
DayOfWeek day = dat.DayOfWeek;
MessageBox.Show(day.ToString());        // zeigt "Thursday"
```

Leider wird in obigem Beispiel nur der englische Wochentag angezeigt. Um den deutschen Bezeichner zu erhalten, können Sie die (statische) *Format*-Methode der *String*-Klasse verwenden (siehe Abschnitt 4.6.2).

BEISPIEL 4.91: Der wievielte Tag im Jahr ist der 23.8.2012?

C#
```
DateTime dat = Convert.ToDateTime("23.8.2012");
int i = dat.DayOfYear;                  // i erhält den Wert 236
```

4.4.3 Wichtige Methoden von DateTime-Variablen

Methode	Beschreibung
AddDays(double *val*)	... liefert ein neues Datum zurück, welches sich aus der Addition des übergebenen Wertes zum Datum berechnet. *val* ist eine Zahl, die aus ganzen Tagen und den Bruchteilen eines Tages besteht, sie kann auch negativ sein!
AddHours(double *val*)	
AddMinutes(double *val*)	
AddMonths(double *val*)	
AddYears(doble *val*)	
ToLongDateString()	... liefern das Datum als kurzen oder langen Datums-/Zeitstring und können als Alternative zur *ToString*-Methode eingesetzt werden.
ToLongTimeString()	
ToShortDateString()	
ToShortTimeString()	

BEISPIEL 4.92: Das Datum, was genau 2,5 Tage vor dem 23.8.2012, 16:30, liegt

C#
```
DateTime dat = Convert.ToDateTime("23.8.2012 16:30");
dat = dat.AddDays(-2.5);
MessageBox.Show(dat.ToString());       // zeigt "21.08.2012 04:30:00"
```

BEISPIEL 4.93: Formatierte Ausgabe der Zeit

C#
```
DateTime dat = Convert.ToDateTime("23.8.2012 16:30");
MessageBox.Show(dat.ToShortTimeString());      // zeigt "16:30"
```

HINWEIS: Zahlreiche weitere Möglichkeiten der Formatierung von *DateTime*-Variablen finden Sie im Abschnitt 4.6.1.

4.4.4 Wichtige Mitglieder der DateTime-Struktur

Eigenschaft/Methode	Erläuterung
Now	... liefert das aktuelle Datum mit Zeit
Today	... liefert aktuelles Datum ohne Zeit
DaysInMonth(int *year*, int *month*)	... liefert die Anzahl von Tagen eines Monats
IsLeapYear(int *year*)	... stellt fest, ob das Jahr ein Schaltjahr ist
Parse(string *s*)	... dient zum Umwandeln eines Strings in einen *DateTime*-Wert mit optionalen Kulturinformationen

BEISPIEL 4.94: Die Tage des Monats Februar 2012

C#
```
int i = DateTime.DaysInMonth(2012, 2);          // i erhält den Wert 29
```

BEISPIEL 4.95: Ist das Jahr 2012 ein Schaltjahr?

C#

```
int year = 2012;
string s = "Das Jahr " + year.ToString() + " ist kein Schaltjahr!";
if (DateTime.IsLeapYear(year))
          s = s.Replace("kein", "ein");
MessageBox.Show(s);
```

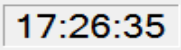
Ergebnis

BEISPIEL 4.96: Eine Uhr anzeigen

C#

Bestücken Sie ein Formular mit einem *Label* und einem *Timer*-Steuerelement (*Interval* = 1000, *Enabled* = *True*). Erzeugen Sie durch Doppelklick auf den *Timer* den Rahmencode für den *Tick*-Eventhandler:

```
private void timer1_Tick(object sender, EventArgs e)
{
     label1.Text = DateTime.Now.ToString("HH:mm:ss");
}
```



4.4.5 Konvertieren von Datumstrings in DateTime-Werte

In Eingabemasken werden Datumswerte häufig als Zeichenketten eingetippt (z.B. in *TextBox*). Deshalb ist nachfolgend eine explizite Konvertierung in den *DataTime*-Typ erforderlich.

BEISPIEL 4.97: Datum und Zeit werden in jeweils eine *TextBox* eingegeben, in einen *DateTime*-Typ konvertiert und in einem *Label* angezeigt

C#

```
DateTime d = Convert.ToDateTime(textBox1.Text + " " + textBox2.Text);
label1.Text = d.ToString("dd.MM.yy HH:mm");
```

Ergebnis

Als Alternative zur *Convert*-Klasse lässt sich auch mit der (statischen) *Parse*-Methode ein *string* in einen *DateTime*-Wert umwandeln[1].

BEISPIEL 4.98: Eine äquivalente Realisierung des Vorgängerbeispiels

C#
```
DateTime d = DateTime.Parse(textBox1.Text + " " + textBox2.Text);
...
```

4.4.6 Die TimeSpan-Struktur

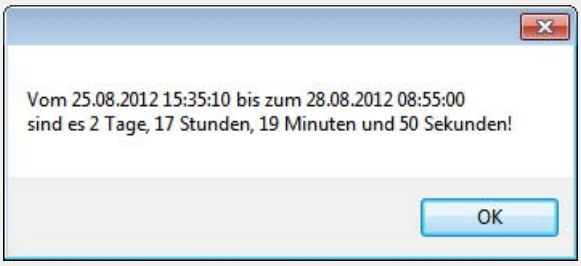
Im Unterschied zum *DateTime*-Typ, der einen konkreten Zeitpunkt beschreibt, bezieht sich der Typ *TimeSpan* auf eine bestimmte Zeitspanne. Der Wert einer *TimeSpan*-Variablen entspricht der Anzahl von Ticks.

Verschiedene Konstruktoren nehmen Tage, Stunden, Minuten, Sekunden usw. als Parameter entgegen, aber auch Ticks.

BEISPIEL 4.99: Die Zeitdauer zwischen zwei Zeitpunkten wird berechnet und in Tagen, Stunden, Minuten und Sekunden aufgeschlüsselt

C#
```
DateTime d1 = new DateTime(2012, 8, 25, 15, 35, 10);    // 25.8.2012 15:35:10
DateTime d2 = new DateTime(2012, 8, 28, 8, 55, 0);      // 28.8.2012 08:55:00

TimeSpan ts = new TimeSpan(d2.Ticks - d1.Ticks);

 string tg = ts.Days.ToString();
 string std = ts.Hours.ToString();
 string min = ts.Minutes.ToString();
 string sek = ts.Seconds.ToString();

 MessageBox.Show("Vom " + d1.ToString() + " bis zum " + d2.ToString() +
             "\nsind es " + tg + " Tage, " + std + " Stunden, " + min.ToString() +
             " Minuten und " + sek.ToString() + " Sekunden!");
```

Ergebnis



Wie Sie obigem Beispiel entnehmen, lassen sich Tage, Stunden usw. als Eigenschaften eines *TimeSpan* auslesen.

[1] Dies hat den Vorteil, dass zusätzlich Kulturinformationen mit übergeben werden dürfen, welche die Besonderheiten des Kalenders eines bestimmten Landes berücksichtigen (Beispiele siehe Hilfe-Dokumentation).

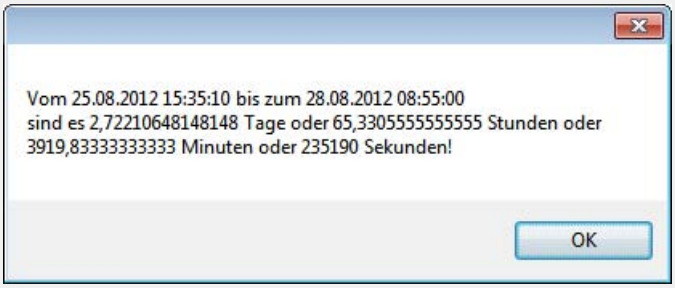
Andere Eigenschaften, die mit *Total...* beginnen, erlauben hingegen eine Auswertung der Gesamtstunden, -minuten etc. zwischen zwei Zeitpunkten.

BEISPIEL 4.100: Modifiziertes Vorgängerbeispiel mit ausgetauschten Eigenschaften (Days => TotalDays, Hours => TotalHours usw)

C#

```
  ...
  string tg = ts.TotalDays.ToString();
  string std = ts.TotalHours.ToString();
  string min = ts.TotalMinutes.ToString();
  string sek = ts.TotalSeconds.ToString();
...
```

Ergebnis



Eine nützliche Anwendung von *TimeSpan* zur Zeitmessung im Millisekundenbereich finden Sie im Praxisbeispiel

► 4.7.2 Zeichenketten mit StringBuilder addieren

4.5 Mathematische Funktionen

Die *Math*-Klasse stellt eine Vielzahl von (statischen) Methoden zur Verfügung, mit denen Sie die üblichen mathematischen Funktionen aufrufen können.

4.5.1 Überblick

Die folgende Tabelle zeigt eine Zusammenstellung wichtiger Funktionen:

Methode	Erläuterung
`Abs(x)`	liefert den Absolutwert einer Zahl x (Wert ohne Vorzeichen)
`Atan(x)`	berechnet den Arcustangens einer Zahl x
`Cos(x)`	berechnet den Cosinus eines Winkels x (Bogenmaß)
`Exp(x)`	liefert e^x (die x-te Potenz zur Basis e)
`Log(x)`	liefert natürlichen Logarithmus (Basis e)
`Log10(x)`	liefert dekadischen Logarithmus (Basis 10)

Methode	Erläuterung
`Max(x, y)`	liefert Maximum von x und y
`Min(x, y)`	liefert Minimum von x und y
`Pi`	die Zahl Pi = 3,1415....
`Pow(x, y)`	berechnet die Potenz x^y
`Round(x,n)`	rundet den Ausdruck x auf n Nachkommastellen
`Sign(x)`	berechnet das Vorzeichen eines Ausdrucks x
`Sin(x)`	berechnet den Sinus eines Winkels x (Bogenmaß)
`Sqrt(x)`	ermittelt Quadratwurzel von x
`Tan(x)`	berechnet Tangens eines Winkels x (Bogenmaß)

HINWEIS: Die zu übergebenden Argumente und der Rückgabewert der Methoden der *Math*-Klasse haben im Allgemeinen den *double*-Datentyp. Es sind deshalb die entsprechenden Typkonvertierungen zu bemühen (siehe Kapitel 2).

4.5.2 Zahlen runden

BEISPIEL 4.101: Rundung einer *float*-Zahl auf zwei Nachkommastellen.

C#
```
float f = (float) (Math.Round(12.477, 2));
MessageBox.Show(f.ToString());              // zeigt "12,48"
```

4.5.3 Winkel umrechnen

Der Winkel wird grundsätzlich im Bogenmaß (*Radian*) angegeben. Die Umrechnungsformel lautet:

Rad = Grad * Pi /180

BEISPIEL 4.102: Der Sinus von 30 Grad

C#
```
double d = Math.Sin(30 * Math.PI / 180);    // liefert 0,5
```

4.5.4 Potenz- und Wurzeloperationen

Für diese Operationen steht die *Pow*-Methode der *Math*-Klasse zu Diensten.

BEISPIEL 4.103: Die dritte Wurzel aus 10

C#
```
double d = 10;
d = Math.Pow(10, 1 / 3.0);
MessageBox.Show(d.ToString("0.0000"));      // zeigt 2,1544
```

4.5.5 Logarithmus und Exponentialfunktionen

Log(x) und *Exp(x)* sind zueinander Umkehrfunktionen. Sie beziehen ihre Basis auf die Zahl *e* (2,718282...). Es gilt daher der Zusammenhang:

```
Exp(Log(x)) = Log(Exp(x)) = x
```

BEISPIEL 4.104: Der "Beweis" für obige Formel[1]

C#
```
const double x = 25;
double y;
y = Math.Exp(Math.Log(x));
MessageBox.Show(y.ToString());    // zeigt "25"
y = Math.Log(Math.Exp(x));
MessageBox.Show(y.ToString());    // zeigt ebenfalls "25"
```

BEISPIEL 4.105: Nachbildung des dekadischen Logarithmus (zur Basis 10) durch eine selbst definierte Methode

C#
```
private double Log10(double x)              // Methodendeklaration
{
   return(Math.Log(x) / Math.Log(10));
}
double d = Log10(100);            // Aufruf der Methode
MessageBox.Show(d.ToString());    // zeigt "2"
```

Da nicht nur der natürliche, sondern auch der dekadische Logarithmus von der *Math*-Klasse zur Verfügung gestellt wird, ist man auf den "Eigenbau" der *Log10*-Funktion nicht mehr angewiesen.

BEISPIEL 4.106: Das Vorgängerbeispiel wird vereinfacht.

C#
```
double d = Math.Log10(100);
MessageBox.Show(d.ToString());     // zeigt "2"
```

4.5.6 Zufallszahlen erzeugen

Zum Erzeugen von Pseudozufallszahlen verwenden Sie die *Random*-Klasse. Im Konstruktor können Sie einen Startwert übergeben, der für jeweils unterschiedliche Folgen sorgt. Der *Next*-Methode übergeben Sie als Argumente den unteren und oberen Grenzwert, dabei gilt:

untere Grenze <= Zufallszahl < obere Grenze

BEISPIEL 4.107: Es werden ganze Zufallszahlen zwischen 0 und 9 erzeugt.

C#
```
Random rdm = new Random((int) DateTime.Now.Ticks);    // Initialisieren mit Systemzeit
int z = rdm.Next(0, 10);
```

[1] Den Mathematikpuristen unter den Lesern werden hier möglicherweise die Haare zu Berge stehen, aber hier geht es nur um reine Programmierpraxis.

BEISPIEL 4.107: Es werden ganze Zufallszahlen zwischen 0 und 9 erzeugt.

```
MessageBox.Show(z.ToString());        // zeigt Zufallszahl aus 0,1,2 ... 9
```

HINWEIS: Die Methode *NextDouble* liefert eine Zufallszahl zwischen 0,0 und 1,0.

4.6 Zahlen- und Datumsformatierungen

Um Daten in Ihrem Programm anzuzeigen, müssen diese meist vorher in den *string*-Datentyp verwandelt werden. In den meisten Fällen brauchen Sie für die Datenanzeige die hohe Genauigkeit nicht, mit denen C# die Zahlenwerte berechnet hat, oder aber Sie wünschen einfach einige "kosmetische" Korrekturen. Die beiden wichtigsten Möglichkeiten, um sich diese Wünsche zu erfüllen, sind:

- Anwenden der *ToString*-Methode der Variablen,
- Anwenden der *Format*-Methode der *String*-Klasse.

Beide Verfahren lassen sich nicht nur zum Formatieren von Zahlen, sondern auch für Datums-/Zeitangaben verwenden.

HINWEIS: Bei Formatierungen werden die Originalwerte grundsätzlich nicht beeinflusst!

4.6.1 Anwenden der ToString-Methode

Alle Datentypen – mit Ausnahme von *string* – verfügen über die *ToString()*-Methode, welche nicht nur einen beliebigen Wert in einen *string* konvertiert, sondern ihn auch noch formatieren kann.

SYNTAX: *var*.ToString(*formatString*)

Bereits im Praxisbeispiel 1.7.2 hatten wir von folgender formatierten Anzeige Gebrauch gemacht:

BEISPIEL 4.108: Anzeige einer Gleitkommazahl mit mindestens einer Vorkomma- und zwei Nachkommastellen

C#
```
private float euro;
//...
textBox1.Text = euro.ToString("#,##0.00");
```

Für viele übliche Formatierungen können Sie aber auch die bereits vorhandenen standardisierten Formatstrings einsetzen.

BEISPIEL 4.109: Wir verwenden die Standardformatierung "C", die sich auf das in der Systemsteuerung eingestellte Währungsformat bezieht

C#
```
textBox1.Text = euro.ToString("C");
```

Zahlenformatierungen

Die folgende Tabelle zeigt wichtige Zahlenformatierungen:

Formatstring	Beschreibung	Beispielwert	Ergebnis
C	Währung mit Standard-Dezimalstellen	12345.67	12.345,67 €
E	Wissenschaftliches Format	12345.67	1,234567E+004
F2	Zahl mit zwei festen Dezimalstellen	12345.67	12345,67
F4	Zahl mit vier festen Dezimalstellen	12345.67	12345,6700
G	Allgemeines Format	12345.6789	12345,6789
X	Hexadezimales Format	255	FF
0.00	Nutzerdefiniertes Format	12345.6789	12345,68
#,#0.00	dto.	0.6789	0,68
#,#.00	dto.	0.6789	,68

Datumsformatierungen

Zur Formatierung von Datum und Zeit stehen mit der *ToString*-Methode vielfältige Möglichkeiten zur Verfügung. Auch hier haben Sie die Auswahl zwischen verschiedenen Standardformaten oder benutzerdefinierten Formatierungen:

Formatstring	Beschreibung	Beispiel
d	kurzes Datumsformat	24.08.2012
D	langes Datumsformat	Freitag, 24.August 2012
f	langes Datum mit kurzer Zeit	Freitag, 24.August 2012 08:39
F	langes Datum mit langer Zeit	Freitag, 24.August 2012 08:39:05
g	kurzes Datum mit kurzer Zeit	24.08.2012 08:39
G	kurzes Datum mit langer Zeit	24.08.2012 08:39:05
`m` oder `M`	Monat-/Tag-Format	24 August
t	kurzes Zeitformat	08:39
T	langes Zeitformat	08:39:05
u	Universelles (ISO-)Format	2012-08-24 08:39:05Z
`y` oder `Y`	Jahr Monat	August 2012
dd.MM.yy	nutzerdefiniert	24.08.12
yyyy-MM-dd	dto.	2012-08-24
MM\/dd\/yy	dto.	08/24/12
hh mm:ss	dto.	08:39:05
h:mm	dto.	8:39

BEISPIEL 4.110: Das aktuelle Datum wird in einem Meldungsfenster angezeigt.

C#
```
DateTime d = DateTime.Now.Date;
MessageBox.Show(d.Date.ToString("D"));
```

Ergebnis

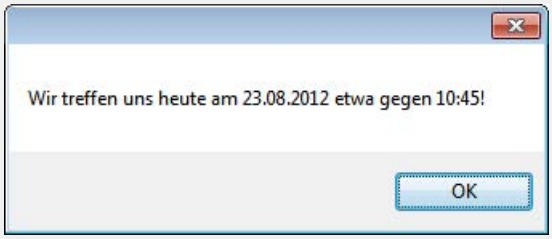
4.6.2 Anwenden der Format-Methode

Hier handelt es sich um eine statische Methode der *String*-Klasse (siehe 4.2.3), mit welcher man recht komplexe Formatierungen erzielen kann.

BEISPIEL 4.111: Zwei Datumsformatierungen sind im Text eingeschlossen.

C#
```
DateTime d = DateTime.Now.Date;
string s = "Wir treffen uns heute am {0:d} etwa gegen {1:h:mm}!";
MessageBox.Show(String.Format(s, d, "10:45"));
```

Ergebnis



Zu obigem Beispiel gibt es sicherlich allerhand Klärungsbedarf:

Der String *s* enthält in den geschweiften Klammern zwei Formatspezifikationen, deren allgemeine Syntax wie folgt ist:

SYNTAX: `{Index[, Minimalbreite][:Formatstring]}`

- Der **Index** bestimmt das zu formatierende Argument. In unserem Beispiel gibt es zwei Argumente, nämlich d und #10:45 AM#, das erste hat (wie immer) den Index 0, das zweite den Index 1.
- Mit **Minimalbreite** können Sie optional angeben, wie breit der formatierte Teilstring mindestens sein soll. Für positive Werte wird der String nach links, für negative Werte nach rechts mit Leerzeichen aufgefüllt.

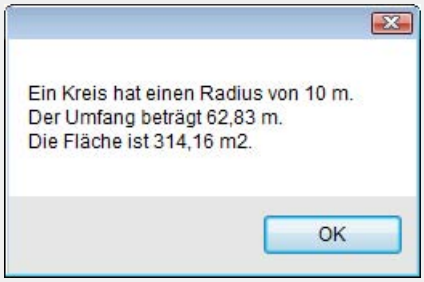
- Der optionale **Formatstring** wird nach denselben Regeln erstellt, wie wir sie bereits im vorhergehenden Abschnitt kennen gelernt haben.

Es folgt ein weiteres Beispiel aus dem mathematischen Bereich, bei welchem drei Argumente berechnet und formatiert werden.

BEISPIEL 4.112: Die Kenngrößen eines Kreises werden ermittelt.

C#
```
int r = 10;
const double Pi = Math.PI;            // statische Eigenschaft der Math-Klasse
const char CR = (char) 13;            // Zeichen für Zeilenumbruch
string s = "Ein Kreis hat einen Radius von {0} m." + CR
         + "Der Umfang beträgt {1:0.00} m." + CR
         + "Die Fläche ist {2:0.00} m2.";
MessageBox.Show(String.Format(s, r, 2 * Pi * r, Pi * r * r));
```

Ergebnis



Zum Schluss noch ein ganz einfaches Beispiel, welches auch die letzten Unklarheiten beseitigen dürfte.

BEISPIEL 4.113: Der Wochentag des aktuellen Datums wird angezeigt

C#
```
label1.Text = String.Format("{0:dddd}", DateTime.Now);      // liefert z.B. "Dienstag"
```

Wer meint, dass obiges Beispiel mit der *DayOfWeek*-Methode doch viel einfacher zu lösen wäre, der vergisst, dass diese Methode nur den englischsprachigen Tag liefert.

BEISPIEL 4.114: Der englische Wochentag des aktuellen Datums wird angezeigt

C#
```
DateTime dat = DateTime.Now;
label1.Text = dat.DayOfWeek.ToString();           // liefert z.B. "Tuesday"
```

4.7 Praxisbeispiele

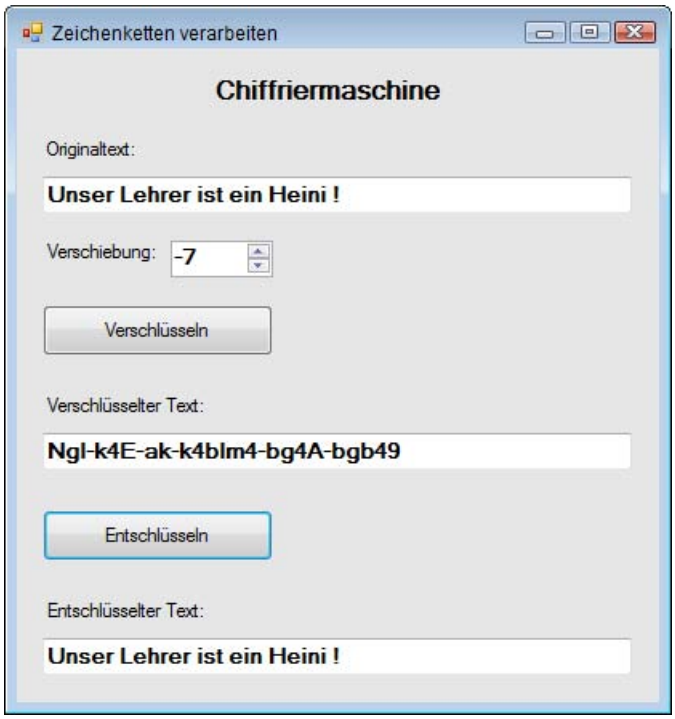
4.7.1 Zeichenketten verarbeiten

Die zahlreichen Eigenschaften und Methoden von Stringvariablen sollen an folgendem, nicht ganz ernst zu nehmenden, Beispiel demonstriert werden:

Vielleicht haben Sie sich in Ihrer Schulzeit schon mal gegenseitig "hochgeheime" Nachrichten nach folgendem Prinzip zugeschickt: Jeder Buchstabe des Originals wird um eine bestimmte Anzahl Stellen innerhalb der alphabetischen Reihenfolge verschoben. Dem Empfänger teilt man vorher den "Geheimschlüssel" mit, d.h. die Anzahl der Buchstabenverschiebungen nach rechts bzw. links. Beispielsweise wird der Text "hallo" mit einer Verschiebung von drei Stellen als "kdoor" verschlüsselt. Ausgerüstet mit einer Schablone kann man auf diese Weise relativ schnell kleinere Texte chiffrieren und dechiffrieren.

Oberfläche

Die folgende Laufzeitansicht zeigt die benötigten Steuerelemente (von oben nach unten: *textBox1, numericUpDown1, button1, textBox2, button2, textBox3, button3*):



Die Anpassung der *Font*-Eigenschaft der drei *TextBox*en ist empfehlenswert (*Font.Bold = True*). Die Eigenschaften *Value* und *Increment* der *NumericUpDown*-Komponente belassen wir auf ihren Standardwerten (*0* bzw. *1*).

Quellcode

Die fett gedruckten Elemente im Code weisen auf Eigenschaften bzw. Methoden der *String*-Klasse hin, wie wir sie beispielsweise zum Herauskopieren (*Substring*) bzw. Suchen (*IndexOf*) eines bestimmten Zeichens benötigen.

```
public partial class Form1 : Form
{   ...
```

Zu Beginn deklarieren wir den Zeichensatz als Stringkonstante, in welcher alle erlaubten Buchstaben, Zahlen, Leerzeichen etc. enthalten sein müssen. Die Reihenfolge ist von untergeordneter Bedeutung (siehe Bemerkung am Schluss). Außerdem deklarieren wir eine Variable, welche die Länge des Zeichensatzes ermittelt:

```
   private const string s0 = "abcdefghijklmnopqrstuvwxyz" +
                             "äöüABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ1234567890 .,-?!";
   private int nmax = s0.Length;             // Anzahl der Zeichen
```

Beim Laden des Formulars werden der obere und untere Grenzwert der *NumericUpDown*-Komponente auf die positive bzw. negative Länge des Zeichensatzes gesetzt:

```
   private void Form1_Load(object sender, EventArgs e)
   {
       numericUpDown1.Maximum = nmax;
       numericUpDown1.Minimum = -nmax;
   }
```

Das Verschlüsseln und das Entschlüsseln des Textes wird von einer einzigen Funktion erledigt. Als Übergabeparameter erhält sie einen String *s* sowie die gewünschte (positive oder negative) Verschiebung *n*. Rückgabewert ist der verschlüsselte bzw. der entschlüsselte String:

```
   private string codieren(string s, int n)
   {
       string s1 = String.Empty;      // zurückzugebender String
```

In der folgenden Schleife wird pro Durchlauf ein Zeichen aus dem übergebenen String "herauskopiert" und seine Position im Zeichensatz gesucht. Anschließend werden das verschobene Zeichen berechnet und der Ergebnisstring stückweise wieder "zusammengebaut":

```
       for (int i = 0; i < s.Length; i++)
       {
            char z = s.Substring(i)[0];    // i-tes Zeichen herauskopieren
            int pos = s0.IndexOf(z);       // Position im Zeichensatz suchen
            if (pos ==-1)                  // Zeichen nicht gefunden
            {
                MessageBox.Show(z + " ist ein unzulässiges Zeichen!", "Warnung");
                break;
            }
            int posN = pos + n;           // auf neue Position verschieben
            if (posN >= nmax) posN = posN - nmax;    // bei Überlauf wieder von vorn beginnen
            if (posN < 0) posN = posN + nmax;        // ... bzw. hinten weitermachen
            z = s0.Substring(posN)[0];               // korrespondierendes Zeichen ermitteln
```

```
            s1 = s1 + z;                              // Rückgabestring zusammensetzen
        }
        return(s1);
    }
```

Der Funktionsaufruf beim Verschlüsseln:

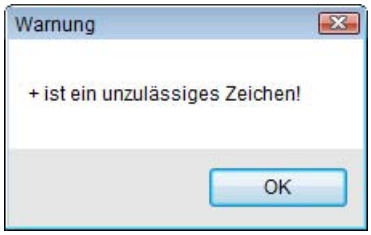
```
    private void button1_Click(object sender, EventArgs e)
    {
        textBox2.Text = codieren(textBox1.Text, (int) numericUpDown1.Value);
    }
```

Der Aufruf beim Entschlüsseln:

```
    private void button2_Click(object sender, EventArgs e)
    {
        textBox3.Text = codieren(textBox2.Text, (int) -numericUpDown1.Value);
    }
}
```

Test

Nach dem Programmstart haben Sie die Möglichkeit zu umfassenden Experimenten. Falls Sie ein nicht erlaubtes Zeichen eingeben, erfolgt ein Hinweis:



Bemerkungen

- Es liegt an Ihnen, den erlaubten Zeichenvorrat zu vergrößern bzw. einzuschränken. Dazu brauchen Sie lediglich die Stringkonstante *s0* zu ändern.
- Die "Knackfestigkeit" des Verfahrens lässt sich deutlich steigern, wenn Sie die Zeichen innerhalb s0 nicht in alphabetischer Reihenfolge, sondern zufällig anordnen.
- Die Achillesferse unserer "Chiffriermaschine" soll nicht verschwiegen werden: Der Hacker sucht im Text zunächst nach dem am häufigsten vorkommenden Zeichen, und das ist mit hoher Wahrscheinlichkeit das verschlüsselte "e". In obiger Laufzeitabbildung kommt deshalb das "-" am häufigsten vor. Fast ist der Code geknackt! Allerdings werden die Tüftler unter Ihnen bald einen Weg finden, wie sich auch dieser Angriffspunkt entschärfen lässt.

4.7.2 Zeichenketten mit StringBuilder addieren

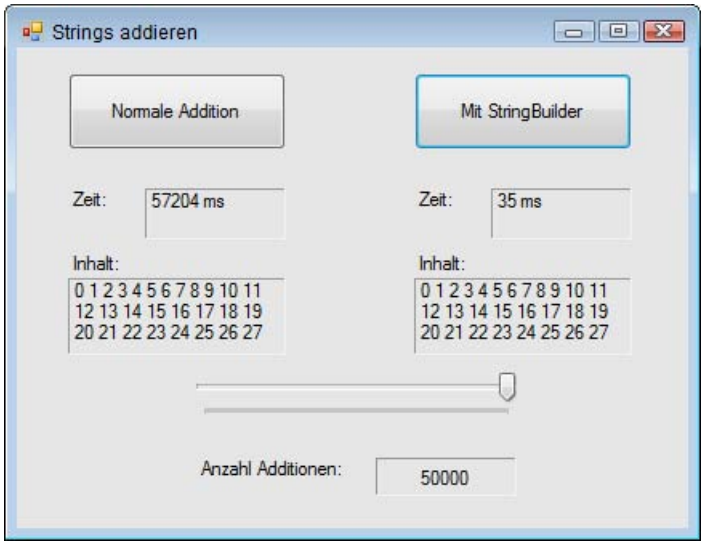
Unter .NET sind auch Strings Referenztypen, d.h., eine Stringvariable speichert nicht den Wert, sondern lediglich einen Verweis (Referenz, Adresse) auf die Speicherstelle. Allerdings muss ein wichtiger Unterschied beachtet werden:

HINWEIS: Strings werden im Speicher als unveränderliche Zeichenketten abgelegt. Mit jeder Änderung einer String-Variablen wird ein neuer String erzeugt!

In diesem Rezept können Sie sich davon überzeugen, was für ein gefährlicher Zeit- und Speicherplatzfresser deshalb die einfache Stringaddition mittels + Operator sein kann und welch gewaltige Performancesteigerung man durch Verwendung eines *StringBuilder*-Objekts erreicht. Nebenbei erfahren Sie auch etwas über den Einsatz der *TimeSpan*-Klasse zur exakten Zeitmessung.

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie eine Benutzerschnittstelle mit zwei *Button*s, diversen *Label*s und einer *TrackBar*. Weisen Sie der *TrackBar* die Werte *Maximum = 50000*, *Minimum = 1000*, *SmallChange = 1000*, *LargeChange = 10000* zu. Die folgende Abbildung zeigt bereits die Laufzeitansicht mit Testergebnissen.



Quellcode Klasse CTest

Über das Menü *Projekt|Klasse hinzufügen...* ergänzen Sie das Projekt um eine neue Datei mit dem Namen *Test.cs*.

```
using System;
using System.Text;
```

...

Die statische Klasse *CTest* stellt zwei Methoden bereit, welche die beiden Verfahren zur Stringaddition demonstrieren sollen. Als Rückgabewert der beiden Methoden dient die Struktur *TestResult*, deren Felder den Ergebnisstring und die zur Addition benötigte Zeit (in Millisekunden) kapseln:

```
public struct TestResult
{
    public string inhalt;
    public int zeit;
}

public static class CTest
{
```

Beide Methoden addieren in einer Schleife (beginnend mit Eins) die ganzen Zahlen fortlaufend zu einem Teststring. Beim Aufruf wird ein Parameter *nr* übergeben, welcher die Anzahl der Schleifendurchläufe bzw. Additionsoperationen festlegt.

Die erste Methode demonstriert die übliche Stringaddition mittels + Operator. Für die Zeitmessung wird die *TimSpan*-Klasse benutzt:

```
    public static TestResult addNormal(int nr)
    {
        DateTime d1 = DateTime.Now;
        TestResult r;
```

String addieren:

```
        string str = String.Empty;
        for (int i = 0; i < nr; i++)
            str += i.ToString() + " ";
        r.inhalt = str;
```

Zeitmessung:

```
        DateTime d2 = DateTime.Now;
        TimeSpan ts = new TimeSpan(d2.Ticks - d1.Ticks);
        r.zeit = Convert.ToInt32(ts.TotalMilliseconds);
        return r;
    }
```

Analog ist die Methode *addWithStringBuilder* aufgebaut, welche zur Addition die *Append*-Methode der *StringBuilder*-Klasse benutzt:

```
    public static TestResult addWithStringBuilder(int nr)
    {
        DateTime d1 = DateTime.Now;
        TestResult r;
```

Mit *StringBuilder* addieren:

```
        StringBuilder sb = new StringBuilder();
```

```
        for (int i = 0; i < nr; i++)
            sb.Append(i.ToString() + " ");
        r.inhalt = sb.ToString();
```

Zeitmessung:

```
        DateTime d2 = DateTime.Now;
        TimeSpan ts = new TimeSpan(d2.Ticks - d1.Ticks);
        r.zeit = Convert.ToInt32(ts.TotalMilliseconds);
        return r;
    }
}
```

Quellcode Form1

```
public partial class Form1 : Form
{
    ...
```

Die normale Stringaddition:

```
    private void button1_Click(object sender, EventArgs e)
    {
        label1.Text = ""; label2.Text = "";
        this.Refresh(); this.Cursor = Cursors.WaitCursor;
        TestResult tr = CTest.addNormal(trackBar1.Value);
        label1.Text = tr.zeit.ToString() + " ms";
        string s = tr.inhalt;
```

Da es sinnlos ist, einen tausende Zeichen langen String komplett in einem *Label* anzuzeigen, beschränken wir uns hier auf die letzten hundert Zeichen:

```
        label2.Text = s.Substring(0, 100);
        this.Cursor = Cursors.Default;
    }
```

Die Addition mittels *StringBuilder*:

```
    private void button2_Click(object sender, EventArgs e)
    {
        label3.Text = ""; label4.Text = "";
        this.Refresh(); this.Cursor = Cursors.WaitCursor;
        TestResult tr = CTest.addWithStringBuilder(trackBar1.Value);
        label3.Text = tr.zeit.ToString() + " ms";
        string s = tr.inhalt;
        label4.Text = s.Substring(0, 100);
        this.Cursor = Cursors.Default;
    }
```

Die Anzeige der Anzahl der Schleifendurchläufe wird aktualisiert:

```
    private void trackBar1_ValueChanged(object sender, EventArgs e)
    {
        label5.Text = trackBar1.Value.ToString();
    }
}
```

Test

Kompilieren Sie das Programm und stellen Sie zuerst die gewünschte Anzahl von Additionen ein. Das Ergebnis ist eindrucksvoll. Wie das oben abgebildete Beispiel zeigt, bringt bei 50000 Durchläufen die Verwendung eines StringBuilders einen Performancegewinn um weit mehr als den Faktor 1000! Allerdings wächst dieses Verhältnis nicht linear. Bei nur 20000 Durchläufen war das Verhältnis immerhin noch 16ms zu 6250ms zugunsten des StringBuilders (Pentium 2,6 GHz).

Bemerkungen

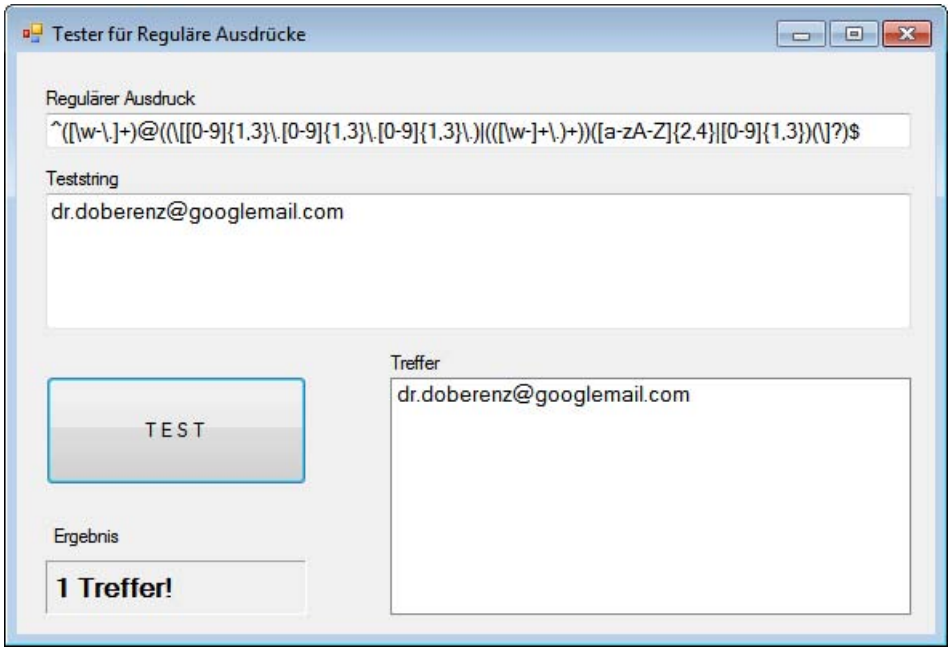
- Da jede Änderung an einer *String*-Variablen einen neuen Wert erzeugt, wird dadurch ein Verhalten quasi wie bei Wertetypen erreicht, obwohl es sich bei Strings – genauso wie z.B. bei Arrays – um Referenztypen handelt.
- Bei der einfachen Stringaddition muss pro Addition stets eine Kopie des Strings im Speicher angelegt werden, selbst wenn nur ein einzelnes Zeichen hinzuzufügen ist. Das ist sehr zeit- und speicherplatzaufwändig.
- Werden, wie in unserem Beispiel, viele Additionen nacheinander ausgeführt, so kommen Sie kaum um die Verwendung eines StringBuilders umhin, wenn Sie Ihr Programm nicht total ausbremsen wollen.
- Wenn Sie beim Instanziieren der *StringBuilder*-Klasse die möglichst gut geschätzte Anfangsgröße des internen Arrays übergeben, können Sie den Code beschleunigen.

4.7.3 Reguläre Ausdrücke testen

Das folgende kleine Programm eignet sich zum Experimentieren mit unterschiedlichsten regulären Ausdrücken und ist als Ergänzung zu den Ausführungen des Abschnitts 4.3 gedacht.

Oberfläche

Die folgende Laufzeitansicht demonstriert die Gültigkeitsüberprüfung einer EMail-Adresse.



Quellcode

```
...
using System.Text.RegularExpressions;
 ...
 public partial class Form1 : Form
 {
     private string rex = null;
     private string text = null;

     private void button1_Click(object sender, EventArgs e)
     {
            rex = textBox1.Text;
            text = textBox2.Text;
            listBox1.Items.Clear();

            MatchCollection rxColl = Regex.Matches(text, rex);
            label1.Text = rxColl.Count.ToString() + " Treffer!";
            foreach (Match m in rxColl)
                listBox1.Items.Add(m.Value);
     }
  }
```

Test

In *textBox1*(oben) geben Sie den regulären Ausdruck ein, in die darunter liegende *textBox2* (*MultiLine=true*) einen beliebigen Teststring. Das *Label* (unten links) zeigt die Anzahl der Treffer, in der *ListBox* daneben werden die einzelnen Treffer zeilenweise aufgeführt.

Bemerkungen

- Auch dem erfahrenen C#-Programmierer wird es zunächst einige Mühe bereiten, die teilweise recht kryptischen Strukturen regulärer Ausdrücke zu durchschauen, muss er dabei doch (quasi en passant) eine neue Sprache erlernen.
- Im Internet finden sich ganze Bibliotheken regulärer Ausdrücke unterschiedlichster Komplexität, die sich als Lernbeispiele eignen.
- Ohne Fleiß kein Preis! Beim Experimentieren werden Sie schnell Folgendes feststellen:

HINWEIS: Reguläre Ausdrücke sind eigentlich relativ einfach zu schreiben, lesen und verstehen hingegen kann sehr mühsam werden!

Eine weitere interessante Anwendung finden Sie im Praxisteil von Kapitel 7!

4.7.4 Methodenaufrufe mit Array-Parametern

Für eine Liste von Zahlen wollen wir *arithmetischen Mittelwert*, *geometrischen Mittelwert* und *Wert in der Mitte* berechnen. Auch in diesem Beispiel geht es weniger um Mathematik als um das Festigen grundlegender Sprachkonzepte.

Oberfläche

Für das Startformular *Form1* benötigen wir an Steuerelementen eine *ListBox*, eine *TextBox*, zwei *Button*s und mehrere *Label*s (siehe Laufzeitansicht am Schluss des Beispiels).

Quellcode

Eine von uns später noch zu definierende Hauptmethode *Average* verwendet drei spezialisierte Methoden: *MeanArithmetic* (für den arithmetischen Mittelwert), *MeanGeometric* (für den geometrischen Mittelwert) und *MeanMedian* (für den Wert in der Mitte der sortierten Folge).

```
public partial class Form1 : Form
{    ...
```

Welche der drei oben genannten Methoden jeweils gemeint ist, wird durch Übergabe einer von drei Konstanten gesteuert, die in einer Enumeration gekapselt sind.

```
    public enum averageType: byte
    {
        MeanA = 0,        // arithmetisches Mittel
        MeanG,            // geometrisches Mittel
```

```
        Median            // Wert in der Mitte
    }
```

Es folgen die Implementierungen der drei Methoden. Allen ist gemeinsam, dass ihnen als Parameter das mit den auszuwertenden Zahlen gefüllte *decimal*-Array übergeben wird.

MeanArithmetic-Funktion: Die Summe aller Zahlen geteilt durch deren Anzahl (arithmetischer Mittelwert).

```
    private decimal MeanArithmetic(decimal[] arr)
    {
        decimal total = 0;
        decimal teiler = arr.GetLength(0);
        if (teiler == 0) return(0);
        int i = 0;
        while (i <= arr.GetUpperBound(0))
        {
            total += arr[i];
            i++;
       }
       return(total / teiler);
    }
```

MeanGeometric-Funktion: Die n-te Wurzel aus dem Produkt aller Zahlen (geometrischer Mittelwert).

```
    private decimal MeanGeometric(decimal[] arr)
    {
        double total=1;
        int i = 0;
        double n = (double) arr.GetLength(0);
        while (i < n)
        {
            total *= (double) arr[i];
            i++;
        }
        return (decimal) Math.Pow(total, 1 / n );    // n-te Wurzel aus total
    }
```

MeanMedian-Funktion: Wenn die Liste eine ungerade Anzahl von Einträgen hat, so ist es der in der Mitte der sortierten(!) Zahlenfolge stehende Wert. Bei einer geraden Anzahl von Einträgen ist es die Summe der beiden mittleren Werte geteilt durch zwei.

```
    private decimal MeanMedian(decimal[] arr)
    {
        int count = arr.GetLength(0);
        if (count == 0) return(0);
        Array.Sort(arr);                    // Zahlen sortieren
        if ((count % 2) == 0)     // für geradzahlige Liste
            return((arr[(int) (count/2.0) - 1] + arr[(int) (count/2.0)])/2);
```

```
        else                        // für ungeradzahlige Liste
            return(arr[count / 2]);      // ganzzahlige Division!
    }
```

Die Hauptmethode *Average* entscheidet auf Grundlage der übergebenen Enumerationskonstanten, zu welcher der drei speziellen Funktionen verzweigt werden soll:

```
    public decimal Average(decimal[] arr, averageType avType)
    {
        switch(avType)
        {
            case averageType.MeanA: return (MeanArithmetic(arr));
            case averageType.MeanG: return (MeanGeometric(arr));
            case averageType.Median: return (MeanMedian(arr));
            default: throw new Exception("Ungültiger Mittelwert-Typ (" + avType.ToString());
        }
    }
```

Nun endlich kommen wir zur Anwendung (Klick auf die "Start"-Schaltfläche):

```
    private void button2_Click(object sender, EventArgs e)
    {
        int n = listBox1.Items.Count;
        decimal[] arr = new decimal[n];   // Array passender Größe deklarieren
        for (int i = 0; i < n; i++)
            arr[i] = Convert.ToDecimal(listBox1.Items[i]);     // Array aus ListBox füllen
```

Dreimaliger Aufruf der *Average*-Funktion mit unterschiedlichen Konstanten:

```
        label1.Text = Average(arr, averageType.MeanA).ToString("#,##0.###");  // MeanArithmetic
        label2.Text = Average(arr, averageType.MeanG).ToString("#,##0.###");  // MeanGeometric
        label3.Text = Average(arr, averageType.Median).ToString();            // Median:
    }
```

Um eine Zahl zur *ListBox* hinzuzufügen, geben Sie diese in die *TextBox* ein und schließen mit der *Enter*-Taste ab:

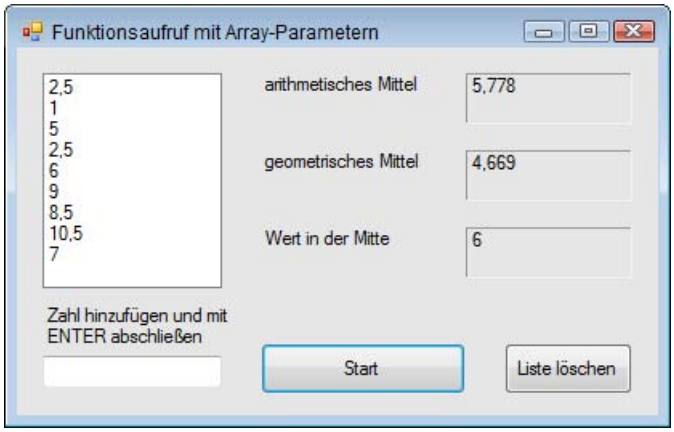
```
    private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        if (e.KeyCode == Keys.Enter)
        {
            listBox1.Items.Add(textBox1.Text); textBox1.Clear();
        }
    }
```

Die *ListBox* löschen:

```
    private void button1_Click(object sender, System.EventArgs e)
    {  listBox1.Items.Clear();  }
}
```

Test

Nach dem Programmstart übertragen Sie zunächst in beliebiger Reihenfolge einige Zahlenwerte aus der *TextBox* in die *ListBox*, um anschließend die Auswertung vornehmen zu können:



Bemerkungen

- Verwenden Sie zur Zahleneingabe das Dezimalkomma und nicht den Dezimalpunkt.
- Jede Eingabe ist mit *Enter* abzuschließen.

Kapitel 5

Weitere Sprachfeatures

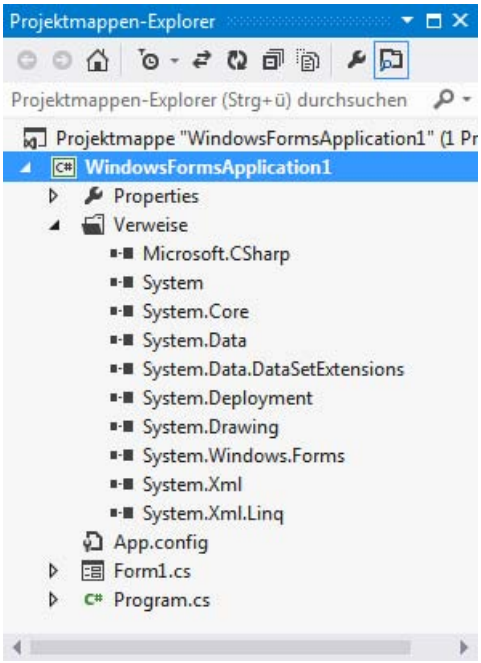
Da Sie nun mit den wichtigsten Sprachelementen und OOP-Konzepten vertraut sind, können wir jetzt die sprachlichen Grundlagen von C# weiter ausbauen und Ihnen weiteres Rüstzeug, wie den Umgang mit Namespaces, Operatorenüberladung, Generics, Collections, Delegates ... mit auf den Weg geben. Auch die dynamische Programmierung soll nicht vergessen werden.

5.1 Namespaces (Namensräume)

Namensräume bilden eine hierarchisch organisierte Verwaltungsstruktur für die inzwischen fast astronomisch große Anzahl von Klassen des .NET Frameworks. Andererseits wäre das Chaos vorprogrammiert, denn die Übersicht ginge verloren und Namenskonflikte ließen sich nicht vermeiden.

5.1.1 Ein kleiner Überblick

Namensräume sind die Kategorien für die Klassendefinitionen, man kann sie näherungsweise mit der Ordnerstruktur eines Dateisystems vergleichen. Demzufolge wird auch jeder Programmcode in Namespaces verwaltet. Eröffnen Sie ein neues Projekt, so wird diesem automatisch von der IDE ein neuer Namensraum zugeordnet (entspricht standardmäßig dem Namen der Projektdatei bzw. Assemblierung). Damit zu Beginn bereits eine gewisse Grundfunktionalität vorhanden ist, werden standardmäßig bereits die wichtigsten Klassen über ihre Namespaces eingebunden. Die entsprechende Liste findet sich im Projektmappen-Explorer, nachdem Sie den Knoten *Verweise* aufgeklappt haben:



Wenn Sie für Ihre Programmentwicklung noch weitere Klassen benötigen, so öffnen Sie das Kontextmenü *Verweise* im Projektmappen-Explorer und wählen *Verweis hinzufügen...*.

5.1.2 Einen eigenen Namespace einrichten

Sie können durchaus den von der Entwicklungsumgebung für Ihr neues Projekt automatisch vergebenen Namespace ändern, allerdings dürfen Sie dies nicht nur an einer Stelle tun, sondern müssen die Namensräume aller zum Projekt gehörenden Dateien bzw. Klassen entsprechend anpassen.

BEISPIEL 5.1: Namespace

C#

Ein neues WindowsForms-Projekt erhält standardmäßig von der IDE den Namen *WindowsFormsApplication1*. Folglich wird es automatisch in den Namespace *WindowsFormsApplication1* eingebunden. Um diesen Namespace in *MyApplication1* zu ändern, müssen Sie die Namespace-Bezeichner in den Quellcode-Dateien *Form1.cs*, *Form1.Designer.cs* und *Program.cs* anpassen.

HINWEIS: Sie können aber auch neue eigene Namespaces eröffnen, das natürlich nicht innerhalb, sondern nur außerhalb von Klassendeklarationen. Das Einbetten in einen anderen Namespace ist hingegen möglich.

BEISPIEL 5.2: Ein eigener Namespace *myNamespace* wird innerhalb des standardmäßig vergebenen Namespace *WindowsFormsApplication1* eingerichtet.

C#
```
namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
    namespace myNamespace
    {
        public class myClass
        { ... }
    }
}
```

HINWEIS: Ein Typ in einem übergeordneten Namespace hat nicht automatisch Zugriff auf einen Typ im untergeordneten Namespace!

BEISPIEL 5.3: Ein Typ im Namespace

C#
```
System.Collections
```
hat keinen Zugriff auf einen Typ im Namespace
```
System.Collections.Generic
```

5.1.3 Die using-Anweisung

Klassen, die sich im gleichen Namespace befinden, "kennen" sich gegenseitig und benötigen deshalb keine voll qualifizierte Namensangabe. Im anderen Fall müssen dem Namen einer Klasse die Namespaces vorangestellt werden, was oft zu langen und unübersichtlichen Ausdrücken führt. Um derartige "Bandwürmer" zu vermeiden, steht am Anfang eines Projekts (außerhalb der Klassendefinitionen) in der Regel eine mehr oder weniger lange Liste von *using*-Anweisungen:

BEISPIEL 5.4: Die *using*-Anweisungen zu Beginn einer neuen Windows Forms-Anwendung

C#
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Dieser einmalige Aufwand führt im nachfolgenden Code zu einer verkürzten und übersichtlichen Typnotation, welche meist nur noch den Typbezeichner erforderlich macht.

BEISPIEL 5.5: Die voll qualifizierte Typdeklaration einer generischen Liste

C#
```
System.Collections.Generic.List<string> myList;
```
... kann wie folgt verkürzt werden:
```
List<string> myList;
```

Das Haar in der Suppe: Wenn in verschiedenen Namensräumen Klassen mit gleichen Typbezeichnern existieren, so führt die Verwendung von *using* zu Mehrdeutigkeiten, denn der Compiler "weiß" nicht, welcher Typ denn nun gemeint ist. In einem solchen Falle sollten Sie die voll qualifizierten Typbezeichner beibehalten. Eine elegantere Möglichkeit bietet aber das Verwenden eines Alias.

5.1.4 Namespace Alias

Ausgangspunkt ist die vollständige Syntax der *using*-Anweisung:

SYNTAX: `using [Alias =] Namespace;`

BEISPIEL 5.6: Ein Alias für den Namespace *System.Collections.Generic* vereinfacht die Deklaration einer generischen Auflistung

C#
```
using GColl = System.Collections.Generic;
...
GColl.List<string> myList;
```

Man kann das sogar noch weiter treiben und selbst komplette Klassenbezeichner durch einen Alias ersetzen.

BEISPIEL 5.7: Eine Modifikation des Vorgängerbeispiels

C#
```
using GCollStrList = System.Collections.Generic.List<string>;
...
GCollStrList myList;
```

5.1.5 Namespace Alias Qualifizierer

Seit C# 2005 gibt es einen Operator, bestehend aus zwei Doppelpunkten (::), die zwischen einem Namespace Alias und seiner Zuordnung stehen. Der Alias des Namespace kann *global* lauten. Das führt dazu, dass zunächst im globalen Namespace "nachgeschaut" wird, statt im direkt zugeordneten Namespace.

Die Fähigkeit des Zugriffs auf ein Mitglied im globalen Namespace ist nützlich, wenn ein Mitglied möglicherweise von einem anderen mit gleichem Namen verdeckt wird.

BEISPIEL 5.8: Namespace Alias Qualifizierer in einer Konsolenanwendung

C#
```
using AliasColl = System.Collections;
namespace System
{
    class CTest
    {
        static void Main()
        {
            AliasColl::ArrayList arrList = new AliasColl::ArrayList();     // Alias auflösen
            arrList.Add("Müller");  arrList.Add("Meier");  arrList.Add("Schultze");
            foreach (string name in arrList)
                global::System.Console.WriteLine(name);                    // Alias auflösen
        }
    }
}
```

Natürlich wäre es ziemlich unsinnig, wenn nicht gar lächerlich, wenn Sie – wie im obigen Beispiel – einen eigenen Namespace mit *System* benennen würden. Trotzdem ist es in größeren Projekten durchaus möglich, dass die Namen von Namespaces doppelt auftreten. In solchen Situationen ist ein globaler Namespace Qualifizierer durchaus hilfreich, da er den direkten Zugriff auf den Root-Namespace ermöglicht.

5.2 Operatorenüberladung

Das Überladen von Operatoren ermöglicht es, dass den bereits in C# vorhandenen Standardoperatoren (siehe Kapitel 2) eine beliebige andere Bedeutung zugewiesen wird.

5.2.1 Syntaxregeln

Die Syntax basiert auf dem Schlüsselwort *operator*.

SYNTAX: `public static` *Ergebnistyp* `operator` *Operator* (*Operand1*, *Operand2*)

Obige Syntax gilt für binäre Operatoren (+,-, *, /, ...), die zwei Operanden für die Ausführung der Operation brauchen.

Für unäre Operatoren (++, --, ...) gilt:

SYNTAX: `public static` *Ergebnistyp* `operator` *Operator* (*Operand*)

HINWEIS: Das Schlüsselwort *operator* kann nur zusammen mit dem Modifizierer *public static* verwendet werden!

5.2.2 Praktische Anwendung

Wie bereits erwähnt, können nur die in C# bereits vorhandenen Standard-Operatoren überladen werden. Im folgenden Beispiel erhält der "+"-Operator eine neue Bedeutung.

BEISPIEL 5.9: Zwei gleichgroße eindimensionale *Double*-Arrays sollen mittels "+"-Operator addiert werden.

C#

Die Klasse *CArray* kapselt den Zugriff auf ein internes eindimensionales Array:

```
public class CArray
{
    public static int N;          // Anzahl der Elemente
    private double[] arr;         // 1-dim. Array
```

Der Konstruktor erzeugt das Array in entsprechender Größe:

```
    public CArray(int n)           // Konstruktor
    {
        N = n;
        arr = new double[n];
    }
```

Ein Indexer ermöglicht den bequemen indizierten Zugriff auf die Elemente der Instanzen der Klasse *CArray*:

```
    public double this[int i]
    {
        get { return arr[i]; }
        set { arr[i] = value; }
    }
```

Eine Überladung des "+"-Operators wird definiert. Rückgabewert ist eine neue Instanz von *CArray*:

```
    public static CArray operator + (CArray a1, CArray a2)
    {
        CArray a = new CArray(N);
        for (int i =0; i < N; i++)
                a = a1[i] + a2[i];
        return a;
    }
}
```

Anwendung der Operatorenüberladung bei der Addition zweier Instanzen von *CArray*:

```
private void button1_Click(object sender, EventArgs e)
{
    CArray A1 = new CArray(3);
    A1[0] = 1.5d; A1[1] = 5; A1[2] = -1;
    CArray A2 = new CArray(3);
    A2[0] = 0; A2[1] = -2; A2[2] = 7;
    CArray A3 = A1 + A2;                  // Anwendung des überladenen + Operators
```

BEISPIEL 5.9: Zwei gleichgroße eindimensionale *Double*-Arrays sollen mittels "+"-Operator addiert werden.

```
    for (int i =0; i < CArray.N; i++)
            MessageBox.Show(A3[i].ToString());    //  "1", "3", "6"
}
```

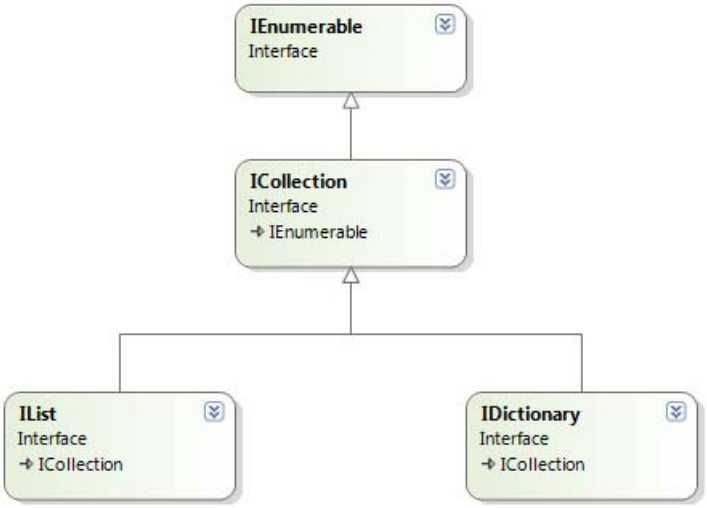
5.3 Collections (Auflistungen)

Wenn Sie mehrere Objekte speichern wollen, so ist es häufig günstiger, dies statt in einem Array (siehe Kapitel 4) in einer Auflistung (Collection) zu tun. Eine Collection verhält sich "intelligenter" als ein Array, es gibt. z.B. keine unbelegten Speicherstellen, wie sie bei einem Array nach dem Löschen eines Eintrags entstehen. Die Größe einer Collection passt sich dynamisch den jeweiligen Erfordernissen an, das Hinzufügen und Löschen von Einträgen wird über spezielle Eigenschaften und Methoden gesteuert. In der Regel ist auch die Performance besser.

Das .NET-Framework stellt zu diesem Zweck über den Namensraum *System.Collections* zahlreiche Auflistungsklassen zur Verfügung, von denen wir hier nur auf zwei wichtige Typen, *ArrayList* und *Hashtable*, näher eingehen wollen.

5.3.1 Die Schnittstelle IEnumerable

Alle Auflistungsklassen verfügen über gemeinsame Methoden, die durch Schnittstellen beschrieben werden, welche von der Schnittstelle *IEnumerable* abgeleitet sind. Die folgende Abbildung zeigt die Beziehungen zwischen diesen Schnittstellen.



Wie Sie erkennen, wird *ICollection* direkt aus *IEnumerable* abgeleitet, wobei beide Schnittstellen die Grundfunktionalität für jede Art von Auflistungsklassen bereitstellen. Nachfolgend zerteilen

sich die Schnittstellen in die Typen *IList* (für Auflistungsklassen auf die per Index zugegriffen wird) und *IDictionary* (für Auflistungen, auf die über eine Schlüssel-Wert-Kombination zugegriffen wird).

IEnumerable

Diese Schnittstelle stellt die Kernfunktionalität einer Collection bereit und macht einen Enumerator verfügbar, der eine einfache Iteration durch eine Auflistung unterstützt.

Die einzige Methode *GetEnumerator* liefert einen Enumerator-Objekt zurück, welches die Methoden *Current*, *MoveNext* und *Reset* hat.

Nach dem Erstellen eines Enumerators oder nach dem Aufrufen der *Reset*-Methode wird ein Enumerator zunächst vor dem ersten Element der Auflistung positioniert. Der erste Aufruf der *MoveNext*-Methode legt den Enumerator auf das erste Element der Auflistung fest.

Wenn *MoveNext* das Ende der Auflistung erreicht, wird der Enumerator hinter dem letzten Element in der Auflistung positioniert, und *MoveNext* gibt *false* zurück. Wenn sich der Enumerator an dieser Position befindet, liefern nachfolgende Aufrufe von *MoveNext* ebenfalls *false*, bis *Reset* aufgerufen wird.

BEISPIEL 5.10: Der Inhalt einer mit String-Objekten gefüllten Auflistung *coll* wird in einer *ListBox* angezeigt

C#
```
IEnumerator en = coll.GetEnumerator();              // Enumerator-Objekt erstellen
while(en.MoveNext())
      listBox1.Items.Add(en.Current);
```

Wie wichtig der im Hintergrund agierende Enumerator ist, wird am Beispiel der *foreach*-Schleife deutlich, mit welcher eine beliebige Auflistung komplett durchlaufen werden kann.

BEISPIEL 5.11: Das Vorgängerbeispiel wird mittels *foreach*-Schleife gelöst

C#
```
foreach (string str in Coll)
            listBox1.Items.Add(str);
```

HINWEIS: Verwechseln Sie bitte nicht *IEnumerable* mit *IEnumerator*. *IEnumerable* ist die Schnittstelle, die von der Auflistungsklasse implementiert wird. *IEnumerator* hingegen ist die Schnittstelle eines Enumerators, wie er von der *GetEnumerator*-Methode der Auflistungsklasse geliefert wird.

ICollection

Diese Schnittstelle erweitert *IEnumerable* um weitere Funktionalität. Hier die wichtigsten Eigenschaften bzw. Methoden:

Member	Beschreibung
Count	... liefert die Anzahl der Elemente in der Auflistung
IsSynchronized	... liefert *true*, wenn Auflistungsobjekt synchronisiert (threadsicher) ist
SyncRoot	... liefert Objektreferenz, die den Objektzugriff synchronisiert
CopyTo()	... kopiert die Elemente der Auflistung in ein Array

IList

Diese Schnittstelle wird von besonders vielen Klassen des .NET-Frameworks implementiert. Hier ihre wichtigsten Mitglieder:

Member	Beschreibung
Item	... ermöglicht den Zugriff auf ein Element der Liste (ist Indexer für *IList*)
Add	... fügt ein Element am Ende der Auflistung hinzu
Clear	... löscht alle Elemente der Auflistung
Contains	... stellt fest, ob ein bestimmtes Element zur Auflistung gehört
IndexOf	... liefert den Index eines bestimmten Objekts
Insert	... fügt ein Objekt an eine bestimmte Position in die Auflistung ein
IsFixedSize	... stellt fest, ob die Kapazität der Auflistung vergrößert werden kann
IsReadOnly	... stellt fest, ob die Auflistung schreibgeschützt ist
Remove	... entfernt ein Element aus der Auflistung unter Angabe der Referenz
RemoveAt	... entfernt ein Element aus der Auflistung unter Angabe des Index

Die *Add*-Methode hat keinerlei Einfluss auf die Positionierung des Elements innerhalb der Liste, denn ein neu hinzugefügtes Element wird immer an das Listenende gesetzt. Der Rückgabewert der *Add*-Methode liefert den Index, den das Element erhalten hat.

HINWEIS: Ein mittels *Remove* oder *RemoveAt* freigegebener Index bleibt nicht unbelegt, sondern bewirkt eine Indexverschiebung aller nachfolgenden Elemente.

5.3.2 ArrayList

Die *ArrayList* ist eine nullbasierte Collection, der mittels *Add*-Methode beliebige Elemente hinzugefügt werden können. Obwohl eigentlich veraltet (nicht generisch, keine *IEnumerable*-Schnittstelle und deshalb kaum für LINQ geeignet), wird die *ArrayList* trotzdem noch häufig eingesetzt.

Beim Erzeugen einer *ArrayList* können Sie eine Startkapazität angeben. Überschreitet die Anzahl der Elemente die Startkapazität, so wird die *ArrayList* automatisch vergrößert. Erzeugen Sie eine *ArrayList* ohne Startkapazität, so wird automatisch der Wert 16 eingestellt.

BEISPIEL 5.12: Drei Nachkommen unterschiedlichen Typs der Klasse *CKunde* werden in einer *ArrayList* (Startkapazität = 3) gespeichert

C#
```
CPrivatKunde kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
CFirmenKunde kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
CFirmenKunde kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");

ArrayList kunden = new ArrayList(3);        // Startkapazität 3
kunden.Add(kunde1);
kunden.Add(kunde2);
kunden.Add(kunde3);
```

Für alle Kunden-Objekte wird die (polymorphe) *addGuthaben*-Methode aufgerufen:

```
foreach (CKunde ku in kundenA)
{
     ku.addGuthaben(brutto);
}
```

HINWEIS: Im Vergleich mit einem normalen Array, das Sie nur ziemlich umständlich dynamisch vergrößern können, schneidet eine *ArrayList* deutlich schneller ab (zum Teil mehr als Faktor 100!).

Weitere Eigenschaften und Methoden

Ein Direktzugriff auf die einzelnen Elemente entspricht dem bei einem normalen Array, allerdings ist dazu explizite Typkonvertierung erforderlich.

BEISPIEL 5.13: Das Guthaben des ersten Kunden wird angezeigt.

C#
```
CKunde kd = (CKunde) kunden[0];                 // Typecasting
label1.Text = kd.guthaben.ToString("C");
```

Mit *AddRange* fügen Sie Elemente aus einer anderen Auflistung hinzu.

BEISPIEL 5.14: Zwei Personen werden der Kundenliste hinzugefügt.

C#
```
ArrayList personen = new ArrayList[2];
kunden.AddRange(personen);
```

Interessant dürften weiterhin die Methoden *Clear* (entfernt alle Elemente), *RemoveAt* (entfernt Element am angegebenen Index), *Sort* (Sortieren) und *BinarySearch* (Suchen in sortierter Auflistung) sein, die so wie bei einem normalen Array funktionieren.

Einen Vergleichstest finden Sie im Praxisbeispiel

► 5.9.1 ArrayList versus generische List

5.3.3 Hashtable

Die *Hashtable* ist eine Auflistung, die das schnelle Auffinden von Objekten erlaubt. Beim Hinzufügen von Elementen muss allerdings von Ihnen ein eindeutiger (also nicht mehrfach vorkommender) Schlüsselwert angegeben werden. Für den Schlüssel sind beliebige Datentypen möglich, meistens nimmt man dafür *int*- oder *string*-Werte. Allerdings wird in der *Hashtable* nicht der Schlüssel gespeichert, sondern dessen so genannter *Hashcode* (eine automatisch ermittelte Integer-Zahl zur eindeutigen Identifizierung des Schlüsselwerts).

BEISPIEL 5.15: Die drei im Vorgängerbeispiel erzeugten Instanzen der Klassen *CPrivatKunde* bzw. *CFirmenKunde* werden mit den Integer-Schlüsseln 101, 102, 103 in einem *Hashtable*-Objekt gespeichert

C#

```
Hashtable kunden = new Hashtable();

kunden.Add(101, kunde1);
kunden.Add(102, kunde2);
kunden.Add(103, kunde3);
```

Für alle Kunden-Objekte wird die *addGuthaben*-Methode aufgerufen:

```
foreach (CKunde ku in kunden.Values)
{
     ku.addGuthaben(brutto);
}
```

Der direkte Zugriff auf das Element mit dem Schlüssel 102:

```
CKunde kd = (CKunde) kunden[102];
label1.Text = kd.guthaben.ToString("C");
```

HINWEIS: Weil Sie die Felder einer *Hashtable* nicht normal indizieren können, ist das Durchlaufen mittels *for*-Schleife nicht möglich, Sie müssen also immer die *foreach*-Schleife verwenden.

5.3.4 Indexer

Sie wissen bereits aus Kapitel 4 wie ein Array deklariert wird und wie man auf die einzelnen Elemente zugreift, indem man deren Index in eckige Klammern einschließt. Die Elemente eines Arrays können beliebige Objekte sein.

BEISPIEL 5.16: Erzeugen eins Arrays *Personen* mit zwei Objekten der Klasse *CKunde*

C#

```
class CPerson
{
    public string Vorname, Nachname;
}
```

BEISPIEL 5.16: Erzeugen eins Arrays *Personen* mit zwei Objekten der Klasse *CKunde*

Ein Array für zwei Objekte vom Typ *CPerson* wird erzeugt:

```
CPerson[] Personen = new CPerson[2];
Personen[0] = new CPerson { Vorname = "Max", Nachname = "Müller" };
Personen[1] = new CPerson { Vorname = "Kurt", Nachname = "Krause" };
```

Der indizierte Zugriff:

```
MessageBox.Show(Personen[0].Nachname);    // zeigt "Müller"
```

C# bietet Ihnen nun die Möglichkeit, eigene Auflistungsklassen so zu definieren, dass deren Objekte quasi wie ein Array indiziert werden können. Dazu ist ein so genannter Indexer erforderlich, dessen Syntax starke Ähnlichkeit zur Implementierung einer Eigenschaft mittels *get*- und *set*-Accessoren aufweist, wobei der Name dem der Klasse entspricht (*this*).

SYNTAX: *Modifikatoren Datentyp* **this***[Parameterliste]*

BEISPIEL 5.17: Auflistungsklasse

C#

Das Vorgängerbeispiel wird wiederholt, allerdings kommt diesmal statt des Arrays eine selbstdefinierte Auflistungsklasse *CPersonen* zum Einsatz, die mit einem Indexer ausgestattet ist und eine *Hashtable* kapselt, in der die einzelnen *CPerson*-Objekte abgelegt werden. Die Klasse *CKunde* ist natürlich ebenfalls erforderlich (siehe Vorgängerbeispiel).

```
...
public class CPersonen
{
    Hashtable ht = new Hashtable();
```

Jetzt wird der Indexer definiert:

```
    public CPerson this[int i]          // Indexer liefert Objekt vom Typ CPerson
    {
        get
        {
            if (ht.Contains(i)) return ((CPerson)ht[i]);
            else return null;
        }
        set
        {
            if (ht.Contains(i)) ht[i] = value;   // überschreiben, falls Schlüssel existiert
            else ht.Add(i, value);               // sonst hinzufügen
        }
    }
 }
```

Der Zugriff auf die Auflistung unterscheidet sich nicht von der Array-Variante, allerdings braucht die Größe der Auflistung vorher nicht bekannt zu sein, und es können beliebige weitere Objekte hinzugefügt werden:

BEISPIEL 5.17: Auflistungsklasse

```
CPersonen Personen = new CPersonen();
Personen[0] = new CPerson {Vorname = "Max", Nachname = "Müller" };
Personen[1] = new CPerson{Vorname = "Kurt", Nachname = "Krause"};
...
MessageBox.Show(Personen[0].Nachname);        // zeigt "Müller"
```

Es steht Ihnen frei, die Klasse *CPersonen* durch weitere Eigenschaften (*Count*, ...) oder Methoden (*Remove,* ...) zu erweitern, die auf die Fähigkeiten der *Hashtable* zurückgreifen.

HINWEIS: Wegen des Indexers darf die Klasse *CPersonen* keine *Item*-Methode haben!

Es ist auch möglich, den Zugriff auf einen der Accessoren zu beschränken.

BEISPIEL 5.18: Zugriff auf einzelne Zelle eines Arrays mittels Indexer.

C#
```
public double this [int row, int col]
{
   get {return _array[row, col];}
   private set {_array[row, col] = value;}          // Schreibzugriff beschränkt
}
```

HINWEIS: Indexer können niemals *static* deklariert werden, weil sich *this* immer auf ein konkretes Objekt bezieht.

5.4 Generics

Generics muten zunächst wie ein Paradoxon an, erlauben sie doch dem Programmierer die typsichere Verwendung von Klassen, die nicht an einen bestimmten Typ gebunden sind. Die Anwendung ist nicht nur auf Klassen beschränkt, sondern auch auf andere .NET-Typen (Strukturen, Schnittstellen, Delegaten, Methoden) übertragbar.

Gemeinsames Merkmal aller Generics ist die Übergabe von variablen Datentypen (Parametrisierung von Klassen- oder Methodendefinitionen). Der Nutzer einer Klasse gibt also die Datentypen vor, die die Klasse intern verarbeiten soll.

HINWEIS: Die variablen Datentypen werden in spitzen Klammern (<*Datentyp,...*>) unmittelbar nach dem Klassen- bzw. Methodenbezeichner angegeben.

BEISPIEL 5.19: Erzeugen einer generischen Liste zum Speichern von Integerzahlen

C#
```
System.Collections.Generic.List<int> gl = new System.Collections.Generic.List<int>();
```

Um die Vorzüge der neuen Generics richtig würdigen zu können, wollen wir uns zunächst noch einmal das Arbeiten mit den "alten" Generics unter .NET 1.x vor Augen halten.

5.4.1 Klassische Vorgehensweise

Vor Einführung der Generics mussten wir unsere Typen immer in bzw. aus ein(em) *System.Object* casten und verloren so alle Vorzüge einer typorientierten Entwicklungsstrategie. So verlangt auch die bekannte *System.Collection.ArrayList* -Klasse in .NET 1.x, dass die Elemente der *ArrayList* als *System.Object* gespeichert werden.

BEISPIEL 5.20: Objekte einer Klasse *CKunde* werden in einer *ArrayList* gespeichert und wieder ausgelesen.

C#
```
public class CKunde
{
            private string _nachName;

            public CKunde(string nachName)
            { this._nachName = nachName;}
            public string NachName
            { get { return this._nachName; } }
}
```

ArrayList erzeugen und Objekte hinzufügen:

```
System.Collections.ArrayList aList = new System.Collections.ArrayList();
aList.Add(new CKunde("Müller"));
aList.Add(new CKunde("Meier"));
```

Beim Zugriff auf den Inhalt der *ArrayList* ist explizites Typecasting erforderlich:

```
for (int i = 0; i < aList.Count; i++)
{    CKunde kd = (CKunde)aList[i];
     listBox1.Items.Add(kd.NachName);
}
```

Miserable Typsicherheit bei den alten Collections

Die "alten" Collections garantieren keinerlei Typsicherheit. So können Sie zu einer Liste beliebige Objekte unterschiedlichen Typs hinzufügen, ohne dass Ihnen dabei der Compiler auf die Finger haut.

BEISPIEL 5.21: Zu unserer obigen Kundenliste lassen sich problemlos auch Zahlen hinzufügen.

C#
```
System.Collections.ArrayList aList = new System.Collections.ArrayList();
aList.Add(new CKunde("Müller"));
aList.Add(44);      // kein Fehler!
```

Das böse Erwachen kommt erst beim Zugriff:

```
foreach(CKunde kd in aList)  listBox1.Items.Add(kd.NachName);      // Fehler!
```

Viel Aufwand für eigene generische Collection-Klassen

Natürlich kann der Profi *IEnumerable* und *IEnumerator* auch selbst implementieren, um eine eigene typsichere Collection für einen gegebenen Typ, z.B. *CKunde*, zu erzeugen. Das kostet allerdings einiges an Arbeit, und Sie fangen damit immer wieder von vorn an, wenn Sie eine weitere typsichere Auflistung benötigen.

Warum uns auch ein Array nicht weiterhilft

Der scheinbar einfachste Weg zu einer typsicheren Collection ist das Erzeugen eines Arrays des gewünschten Typs.

BEISPIEL 5.22: Ein *Kunden*-Array wird erzeugt und mit zwei Elementen initialisiert.

C#
```
CKunde[] kunden = new CKunde[] {new CKunde("Müller"), new CKunde("Meier")};
foreach (CKunde kd in kunden)  listBox1.Items.Add(kd.NachName);
```

Das garantiert zwar Typsicherheit, die Wiederverwendbarkeit des Codes ist aber miserabel und auch das Arbeiten mit einem solchen Array ist umständlich und ziemlich fehleranfällig. Wenn Sie z.B. einen neuen Kunden hinzufügen wollen, müssen Sie zunächst ein temporäres Array erzeugen und alle Kunden dort hineinkopieren, anschließend das alte Array vergrößern, die Kunden dorthin zurück kopieren und dann den neuen Kunden am Ende des Arrays einfügen.

5.4.2 Generics bieten Typsicherheit

Seit Einführung der Generics steht endlich eine Vorlage zum Erzeugen typsicherer Collections für Elemente beliebigen Typs zur Verfügung. Eine solche generische Klasse ist in der Lage, alle wesentlichen Aufgaben zu erfüllen, die wir für unsere Auflistungen brauchen: Hinzufügen, Löschen, Einfügen, etc. Auch andere Collection-Typen, mit denen wir es allgemein zu tun haben, werden berücksichtigt, so wie z.B. *Stack* (First in, Last out) oder eine *Queue* (First in, First out), etc. Im Folgenden soll das Prinzip der Generics erklärt werden.

Einen einfachen generischen Typ deklarieren

Anhand einer sehr einfachen generischen Klasse wollen wir demonstrieren, wie diese als Container für eine Vielzahl anderer Typen verwendet werden kann.

BEISPIEL 5.23: Die Klasse *CStore<T>* kann den Wert eines beliebigen Typs zu speichern.

C#
```
public class CStore<T>
{
  T t;
  public T Val
  {
      get{return t;}
      set{t = value;}
  }
}
```

Der Klassennamen *CStore<T>* ist der erste Hinweis, dass es sich um einen generischen Typ handelt, die spitzen Klammern umschließen den Typen-Platzhalter *T*. Die Variablendeklaration (*T t;*) erzeugt eine Mitgliedsvariable mit dem Typ *T*, bzw. dem generischen Typ, der erst später mit der Konstruktion der Klasse spezifiziert wird.

Einziges öffentliches Klassenmitglied ist die Eigenschaft *Val*, wobei *T* auch hier den generischen Typ der Eigenschaft repräsentiert.

Generischen Typ verwenden

Um eine generische Klasse für einen beliebigen Typen zu verwenden, brauchen Sie nur eine neue Instanz zu erzeugen, wobei Sie den Namen des gewünschten Typs in den Klammern "<>" übergeben.

BEISPIEL 5.24: Anwendung der Klasse *CStore* (oben) mit Anzeige in einer *ListBox*

C#

Für den Datentyp *System.String*:

```
CStore<string> aString = new CStore<string>();
aString.Val = "Hallo";                                  // Hallo
listBox1.Items.Add(aString.Val);
```

Für den Datentyp *CKunde*:

```
CStore<CKunde> kunde = new CStore<CKunde>();
kunde.Val = new CKunde("Müller");
listBox1.Items.Add(kunde.Val.NachName);                 // Müller
```

Obwohl wir in diesem Beispiel ein und dieselbe Klasse verwendet haben um einen String oder einen Kunden zu speichern, bleibt ihr ursprünglicher Typ unverändert. Und das alles hat funktioniert, ohne dass wir *System.Object* verwenden mussten!

5.4.3 Generische Methoden

Eine Methode mit Typ-Parameter ist generisch. Ähnlich wie bei der Klasse wird der Typ in spitzen Klammern angegeben.

BEISPIEL 5.25: Mit dem folgenden Aufruf wird eine Instanz eines *String*-Arrays mit einer Anfangsgröße von 5 erzeugt:

C#

```
string[] arrStr = CreateArray<string>(5);
```

BEISPIEL 5.26: Diese statische Methode erzeugt ein Array des gegebenen Typs *T* mit der gewünschten Größe *size* und liefert es dem Aufrufer zurück.

C#

```
public static T[] CreateArray<T>(int size)
{
    return new T[size];
}
```

BEISPIEL 5.27: Vertauschen von zwei Objekten

C#
```
private void genSwap<T>(ref T a, ref T b)
{
      T tmp = a;
      a = b;
      b = tmp;
}
```

Anwendung:

```
int i1 = 2;
int i2 = 7;
genSwap(ref i1, ref i2);
```

5.4.4 Iteratoren

Es gibt eine typisierte Methode mit dem Namen *GetEnumerator*. Für die Rückgabe der einzelnen Werte wird das Schlüsselwort *yield* in Verbindung mit *return* benutzt.

BEISPIEL 5.28: Eine Klasse *CMonate* implementiert *GetEnumerator*

C#
```
private class CMonate
{
      public IEnumerator<string> GetEnumerator()
      {
            yield return "Januar";
            yield return "Februar";
            yield return "März";
            ...
            yield return "Dezember";
      }
}
```

Bezüglich der Anwendung der Klasse gibt es keinen Unterschied zur bisher üblichen Vorgehensweise mittels *foreach*-Schleife:

```
CMonate monate = new CMonate();
foreach(string s in monate)
     listBox1.Items.Add(s);
```

Wie nicht anders zu erwarten, erscheinen in der *ListBox* untereinander die zwölf Monate.

Mehr zum Einsatz von *yield* finden Sie im Abschnitt 5.7.2 und im Praxisbeispiel

▶ 5.9.2 Generische IEnumerable-Interfaces implementieren

5.5 Generische Collections

Das wohl wichtigste Einsatzgebiet für Generics sind Collections jeglicher Art.

5.5.1 List-Collection statt ArrayList

Als Alternative zur *ArrayList* bietet sich die generische *List* Klasse an. Aus Gründen der Abwärtskompatibilität musste allerdings auf eine Änderung/Erweiterung der im Namespace *System.Collections* bereits vorhandenen Klassen verzichtet werden.

BEISPIEL 5.29: Erzeugung einer generischen *List* für den benutzerdefinierten Typ *CKunde* (Deklaration der Klasse *CKunde* siehe oben)

C#
```
System.Collections.Generic.List<CKunde> kunden = new
System.Collections.Generic.List<CKunde>();
```

Beachten Sie den *Generic*-Namespace innerhalb des *System.Collections* Namespace. Dort werden Sie weitere generische Klassen wie *SortedList,* Dictionary, *Queue*, *Stack*, .. finden. Die *List*-Klasse in diesem Namespace ähnelt der *System.Collections.ArrayList* Klasse, allerdings gibt es einige wichtige Unterschiede.

BEISPIEL 5.30: (Fortsetzung)

C#

Um die Verwendung der *kunden*-Auflistung zu zeigen, wollen wir dreimal eine Schleife durchlaufen, in welcher wir einen neuen Kunden (*Müller1*, *Müller2*, *Müller3*) erzeugen und diesen zu *kunden* hinzufügen:

```
for(int i = 0; i < 3; i++)
{
      CKunde kunde = new CKunde("Müller" + (i+1).ToString());
      kunden.Add(kunde);
}
```

Durch die Collection iterieren und jedes Element in einer *ListBox* ausgeben:

```
foreach(CKunde kd in kunden)
     listBox1.Items.Add(kd.NachName);
```

Eine andere Möglichkeit zur Ausgabe der Liste wäre eine einfache *for* -Schleife.

BEISPIEL 5.31: Die Nachnamen aller Kunden in einer *ListBox* anzeigen

C#
```
for (int i = 0; i < kunden.Count; i++)
    listBox1.Items.Add(kunden[i].NachName);
```

5.5.2 Vorteile generischer Collections

Die Arbeit mit einer generischen *List* unterscheidet sich deutlich von unseren durch die *ArrayList* (siehe obiges Eingangsbeispiel) geprägten Gewohnheiten. Wir brauchen uns keinerlei Gedanken mehr über den Typ von *CKunde* zu machen, weil unsere generische Collection typsicher arbeitet.

Stellvertretend für die anderen generischen Collections sollen die Vorteile der generischen *List*-Klasse hier noch einmal zusammengefasst werden:

- Das Boxing von Wertetypen ist nicht mehr erforderlich, weil *List<int>* im obigen Beispiel jetzt "weiß", welcher Datentyp zu speichern und wie viel Speicher dafür zu reservieren ist.
- Statt eines Laufzeitfehlers erhalten Sie bereits einen Compiler-Fehler, falls Sie einen falschen Datentyp in der *List* speichern wollen.
- Ein Typecasting entfällt, wenn Sie auf die Werte der generischen *List* zugreifen wollen.

5.5.3 Constraints

Ein Constraint wird benutzt, um die vom Generic akzeptierten Typen zu beschränken. Ähnlich der SQL-Syntax wird dazu das der Klassendefinition nachgestellte *where*-Schlüsselwort benutzt.

BEISPIEL 5.32: Der generischen Typ *Dictionary* mit einem *Constraint*

C#
```
public class Dictionary<K, V> where K : IComparable {}
```

Die *where*-Klausel beschränkt den *K*-Typ auf den Typ *IComparable*. Falls *K* das *IComparable*-Interface nicht implementiert, gibt es einen Compiler-Fehler.

Ein anderer Typ eines Constraints ist der Konstruktor-Constraint.

BEISPIEL 5.33: Um keinen Compilerfehler auszulösen, muss *K* zumindest einen Standardkonstruktor haben.

C#
```
public class CTest<K> where K: new() {}
```

Die prinzipielle Funktion dieser generischen Schnittstellen entspricht ihren nicht-generischen Pendants.

5.6 Das Prinzip der Delegates

Die Delegat-Technologie gehört zweifelsfrei zu den tragenden Säulen von .NET. Jeder, der tiefer in .NET eindringen will, muss sich mit dieser Technik anfreunden, bildet sie doch die Basis für andere wichtige .NET-Features, wie zum Beispiel das Ereignismodell, Callbacks oder die Lambda Expressions bis hin zu LINQ.

5.6.1 Delegates sind Methodenzeiger

Der Begriff "Delegate" bedeutet "delegieren" bzw. "weiterleiten". Damit ist bereits das Wesentliche gesagt, denn ein Delegate macht nichts anderes, als einen Methodenaufruf an eine konkrete Methode weiterzuleiten. Man kann also einen Delegate durchaus auch als typisierten Funktionszeiger (Pointer) bezeichnen. Pointer, wie man sie beispielsweise auch von der Sprache C her kennt, sind ein leistungsfähiges Werkzeug, um Algorithmen flexibel zu gestalten.

Pointer und .NET – das müsste Sie eigentlich stutzig machen, denn das passt nicht so recht zusammen. Die althergebrachte Zeigertechnik ist ziemlich unübersichtlich, nicht selten kommt es zu Speicherzugriffsfehlern und damit zu bösen Programmabstürzen. Aus diesem Grund haben die .NET-Entwickler die klassische Pointertechnologie kategorisch ausgeschlossen. Stattdessen wurden – in konsequenter Weiterführung des objektorientierten Paradigmas des .NET-Frameworks -Methodenzeiger in Objekten gekapselt.

HINWEIS: Ein Delegate ist ein Objekt, das den Zeiger auf die Methode eines anderen Objekts enthält.

Damit wird, wie bei jedem anderen Objekt auch, ein Delegate gewissermaßen als Instanz einer Klasse, d h. eines Objekttyps, erzeugt.

Um von vornherein begriffliche Klarheit zu schaffen, wollen wir gleich von Beginn an die Bezeichnungen "Delegate-Typ" und "Delegate-Objekt" sauber voneinander trennen. Ersterer ist vergleichbar mit dem Begriff der Klasse und der zweite mit dem des Objekts (also der Instanz, die auf eine konkrete Methode zeigt).

5.6.2 Einen Delegate-Typ deklarieren

Ein Delegate-Typ beschreibt die Signatur der Methode, auf welche der daraus instanziierte Delegate zeigen soll. Die Deklaration eines Delegate-Typs hat folgendes Format:

SYNTAX: `<Sichtbarkeit> delegate <Rückgabetyp><Name>([<Parameterliste>])`

BEISPIEL 5.34: Ein Delegate-Typ mit dem Namen *OpDlg* für Rechenoperationen mit zwei Integer-Zahlen

C#
```
public delegate long OpDlg(int x, int y);
```

5.6.3 Ein Delegate-Objekt erzeugen

Ein Delegate kann entweder mit einer benannten oder einer anonymen Methode, oder aber auch mit einem Lambda Expression instanziiert werden. Das Verständnis erschließt sich am leichtesten, wenn wir zunächst von der uns bekannten klassischen Technik zum Aufruf von Methoden ausgehen.

BEISPIEL 5.35: Zwei sehr einfachen Methoden, die zur Addition bzw. Multiplikation von Integer-Zahlen dienen:

C#
```
private long add(int x, int y)
{
    return x + y;
}

private long mult(int x, int y)
{
    return x * y;
}
```

Der herkömmliche Aufruf beider Methoden könnte so aussehen:

```
int a = 7;
int b = 3;
long res1 = add(a, b);          // liefert 10
long res2 = mult(a,b);          // liefert 21
```

Die Methoden *add* und *mult* im obigen Beispiel haben die gleiche Signatur, d h., Anzahl und Typ der übergebenen Parameter sowie der Typ des Rückgabewerts sind identisch. Beide Methoden entsprechen deshalb dem gleichen Delegatentyp.

Um für beide Methodenaufrufe Delegatobjekte zu erzeugen, sind folgende Schritte erforderlich:

1. Definition des Delegattypen mittels *delegate*-Schlüsselwort:

```
delegate long opDlg(int x, int y);
```

2. Deklaration von zwei Variablen vom Typ des Delegaten.

```
opDlg addDlg, multDlg;
```

3. Instanziieren der Delegaten mittels *new* (erst hier werden Delegate-Objekte erzeugt, welche auf die Methoden *add()* und *mult()* zeigen). Als Argument wird der Namen der Methode übergeben, die vom jeweiligen Delegaten aufgerufen werden soll:

```
addDlg = new opDlg(add);
multDlg = new opDlg(mult);
```

4. Der Aufruf der im obigen Beispiel implementierten Methoden *add()* und *mult()* mit den Delegatobjekten *addDlg* bzw. *multDlg* sieht jetzt folgendermaßen aus:

```
int a = 7;
int b = 3;
long res1 = addDlg(a, b);          // liefert 10
long res2 = multDlg(a,b);          // liefert 21
```

Das ist ja ziemlich viel Aufwand, werden Sie jetzt denken, und der Vorteil von Delegaten gegenüber dem direkten Methodenaufruf ist nicht so recht ersichtlich. Für dieses einfache Beispiel mögen Sie vielleicht recht haben, aber im weiteren Verlauf werden Sie feststellen, dass sich mit Delegaten wesentlich mehr anstellen lässt.

BEISPIEL 5.36: Verwendung eines Delegate für eine Methode, die zwei Zahlen vergleicht:

C#
```
private bool compA(int x, int y)                    // eine Methode ...
{ if (y < x) return(true); else return(false); }
...
private delegate bool CompDlg(int x, int y);       // ... dafür Delegatentyp deklarieren
...
CompDlg compMethod = new CompDlg(compA);           // ... und Delegaten instanziieren
...
if (compMethod(i, j)) {…}                          // Methodenaufruf
```

Die Instanziierung kann verkürzt werden, indem die Methode dem Delegaten direkt zugewiesen wird. Die fett gedruckte Zeile darf deshalb wie folgt ersetzt werden:

```
CompDlg compMethod = compA;
```

BEISPIEL 5.37: Obiges Beispiel mit einem generischen Delegaten.

C#
```
private bool compA(int x, int y)
{ if (y < x) return(true); else return(false); }
...
private delegate bool CompDlg<T>(T x, T y);
...
CompDlg<int> compMethod = compA;
...
if (compMethod(i, j)) {...}
```

5.6.4 Delegates vereinfacht instanziieren

Wie auch bei normalen Objekten üblich, können das Erzeugen von Objektvariablen und das Instanziieren in einer einzigen Anweisung realisiert werden.

BEISPIEL 5.38: Das Einführungsbeispiel in einer vereinfachten Variante:

C#
```
delegate long opDlg(int x, int y);
```

Die Schritte 2 und 3 können zusammengefasst werden:

```
opDlg addDlg = new opDlg(add);
opDlg multDlg = new opDlg(mult);
```

Sogar auf den *new*-Operator kann verzichtet werden, sodass sich die letzten beiden Anweisungen weiter vereinfachen:

```
opDlg addDlg = add;
opDlg multDlg = mult;
```

5.6.5 Anonyme Methoden

Bei komplexeren Programmen kann der Aufruf benannter Methoden (in unserem obigen Einführungsbeispiel sind das *add()* und *mult()*) durch einen Delegaten schnell zu unübersichtlichem Code führen. Anonyme Methoden schaffen hier Abhilfe, indem der Methodenkörper direkt an die Instanziierung des Delegate-Objekts angefügt wird. Der Code ist nun nicht mehr namentlich mit einem Methodenbezeichner verbunden und wird deshalb als "anonyme" Methode bezeichnet.

BEISPIEL 5.39: Realisierung des Einführungsbeispiels mit anonymen Methoden. Dazu werden die Schritte 2 und 3 zusammengefasst und auf die benannten Methoden *add()* und *mult()* wird verzichtet.

C#
```
opDlg addDlg = delegate(int x, int y)
{
    return x + y;
};

opDlg multDlg = delegate(int x, int y)
{
    return x * y;
};
```

HINWEIS: Bei einer anonymen Methode handelt es sich nicht mehr um eine eigenständige Methode! Vielmehr geht es hier nur um eine vereinfachte Instanziierung eines Delegate-Objekts!

Anonyme Methoden erlauben es, auf das explizite Deklarieren der Methode zu verzichten und stattdessen den Programmcode in Verbindung mit einem passenden Delegaten direkt zuzuweisen. Einsatzmöglichkeiten ergeben sich in erster Linie bei Ereignisbehandlungsroutinen (Eventhandler).

BEISPIEL 5.40: Auswerten des *Click*-Ereignisses eines *Button*s auf herkömmliche Weise

C#
```
this.button1.Click += new EventHandler(this.button1_Click);       // Eventhandler anmelden
...
```

Eventhandler implementieren:

```
private void button1_Click(object sender, EventArgs e)
{
    this.Close();
}
```

BEISPIEL 5.41: Obiges Beispiel mittels einer anonymen Methode

C#
```
this.button1.Click += delegate(object sender, EventArgs e)
{
   this.Close();
};
```

HINWEIS: Sollen jedoch mehrere Ereignisse den gleichen Code ausführen, so muss an Stelle der Implementierung der Aufruf einer Methode stehen.

5.6.6 Lambda-Ausdrücke

Bei den Lambda-Ausdrücken handelt es sich eigentlich um nichts weiter als um funktional erweiterte anonyme Methoden. Der offensichtlichste Unterschied zu anonymen Methoden zeigt sich in der Syntax: auf die Parameterliste folgt ein Pfeil (=>), gefolgt von einer einzelnen Anweisung oder einem Anweisungsblock:

SYNTAX: `(Inputparameter) => {Expressions/Statements}`

BEISPIEL 5.42: Eine Methode zur Multiplikation von zwei Gleitkommazahlen, die mittels Lambda-Ausdruck definiert wird.

C#
```
public delegate double opDeleg(double x, double y);   // Delegate-Typ

opDeleg multDlg = (double x, double y) => x * y;       // Methode als Lambda-Ausdruck zuweisen
...
textBox1.Text = multDlg(5.5, 4.3).ToString();          // Methodenaufruf: 23.65
```

HINWEIS: Lambda-Ausdrücke stellen lediglich eine leistungsfähigere Syntax dar, mit der sich anonyme Methoden schreiben lassen.

BEISPIEL 5.43: Obiges Beispiel auf traditionelle Weise mittels einer anonymen Methode realisiert.

C#
```
public delegate double opDeleg(double x, double y);   // Delegate-Typ

opDeleg multDlg1 = delegate(double x, double y)        // anonyme Methode zuweisen
{
    return x * y;
};
...
textBox1.Text = multDlg(5.5, 4.3).ToString();          // Methodenaufruf: 23.65
```

Die Syntax von Lambda-Ausdrücken ist gewöhnungsbedürftig. Falls der Code rechts neben dem Pfeil aus mehreren Anweisungen besteht, sind diese in geschweifte Klammern einzuschließen.

BEISPIEL 5.44: Lambda-Ausdruck mit Anweisungsblock

C#
```
(int x) =>
{
    if( x < 0) x = -1 * x;
    return x;
}
```

Der Rückgabetyp eines Lambda-Ausdrucks wird durch den Typ des Ausdruckes hinter dem Pfeil (=>) bestimmt. Folglich hat ein Lambda-Ausdruck mit nur einem Methodenaufruf nach dem Pfeil den gleichen Rückgabetyp wie diese Methode. Falls dem Pfeil ein Anweisungsblock folgt, wird der Rückgabetyp durch den Ausdruck nach *return* festgelegt. Fehlt *return* im Anweisungsblock, dann ist der Rückgabetyp des Lambda-Ausdrucks *void*.

BEISPIEL 5.45: Einige Lambda-Ausdrücke, die die Zuordnung des Rückgabetyps veranschaulichen sollen.

C#

Rückgabetyp *void*, da *Console.WriteLine() void* liefert:

```
(int j) => Console.WriteLine(j.ToString());
```

Rückgabetyp *int*, da *j* und *k* vom Typ *int* sind:

```
(int j, int k) => j * k;
```

Rückgabetyp *void*, da *return* Schlüsselwort im Anweisungsblock fehlt:

```
(double x) =>
{
    x = 3.14;
    MessageBox.Show(x.ToString());
}
```

Rückgabetyp *double*, da ein *double* (0.7) zu einem *int* addiert wird und das Ergebnis *double* ist:

```
(int i) => i + 0.7;
```

Rückgabetyp *string*, da nach dem Pfeil ein *string* addiert wird:

```
(int geb) => "Alles Gute zum " + geb + ". !"
```

Genau wie bei anonymen Methoden auch, muss natürlich die Anzahl der Parameter und deren jeweiliger Datentyp mit denen des Delegate, für den der Lambda-Ausdruck angegeben wird, übereinstimmen. Allerdings kann bei Lambda-Ausdrücken auf die Angabe des Typs für die Parameter verzichtet werden, da diese auch vom Kontext her ableitbar sind. Dem jeweiligen Parameter des Lambda-Ausdrucks wird also automatisch der Typ des entsprechenden Parameters des Delegate, für den der Lambda-Ausdruck angegeben wird, zugewiesen. Im folgenden Beispiel kann man dieses Konzept erkennen.

BEISPIEL 5.46: Parametertyperkennung in einem Lambda-Ausdruck

C#

```
public class CPerson
{
 public string Name = "";
 public int Alter = 0;
}

public delegate void PDelegate(CPerson person);
```

BEISPIEL 5.46: Parametertyperkennung in einem Lambda-Ausdruck

Wegen *PDelegate* ist der Parameter *p* vom Typ *CPerson*, Rückgabetyp muss *void* sein:

```
PDelegate dlg1 = ( p ) => Console.WriteLine(p.Name);
```

Bei nur einem Parameter darf man die Klammern bei *(p)* weglassen:

```
PDelegate dlg2 = p => Console.WriteLine(p.Name);
```

Wie bei den Parametern muss natürlich auch der Rückgabetyp des Lambda Ausdrucks mit dem Rückgabetypen des Delegate übereinstimmen (bzw. eine implizite Konvertierung existieren). Demzufolge liefert der Lambda-Ausdruck im obigen Beispiel keinen Rückgabewert, da *void* vom Delegate *PDelegate* entsprechend als "Rückgabewert" vorgegeben ist.

5.6.7 Lambda-Ausdrücke in der Task Parallel Library

Auch beim Schreiben von Code für die neue Task Parallel Library (TPL) bzw. für Parallel LINQ (PLINQ) werden Sie häufig Lambda-Ausdrücke verwenden.Wenn beispielsweise eine Methode wie *Parallel.For* einen *Func* oder *Action* Delegate als Inputparameter entgegennimmt, so können Sie diesen Parameter als Lambda-Ausdruck bereitstellen. Der Compiler erkennt die Variablentypen innerhalb des Lambda Ausdrucks und konvertiert den Lambda Expression in eine Instanz des Delegaten.

Ausführlich gehen wir auf die *Task Parallel Library* und die parallele Programmierung erst im Kapitel 10 ein.

Um Ihnen dennoch einen ersten Eindruck zu vermitteln, sollen zwei Beispiele gegenübergestellt werden.

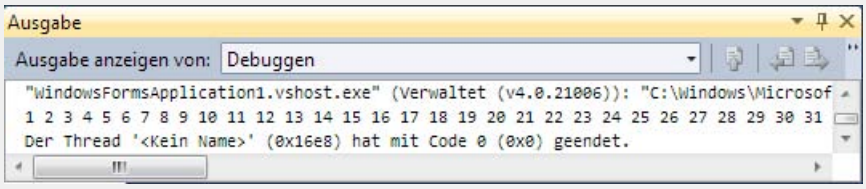
BEISPIEL 5.47: Eine einfache *for*-Schleife in klassischer Programmiertechnik

C#

```
int N = 100;
for (int i = 1; i < N; i++)
      {
          System.Diagnostics.Debug.Write(i.ToString() + " ");
          if (i > 30) break;
      }
```

Ergebnis

Die Ausgabe (Menü *Ansicht|Ausgabe*):



Das analoge Problem soll nun mittels Parallelprogrammierung gelöst werden.

BEISPIEL 5.48: Anwendung eines Lambda-Ausdrucks in der *Parallel.For()* Methode

C#

```
using System.Threading.Tasks;
...
int N = 100;
Parallel.For(0, N, (i, lState) =>
{
    System.Diagnostic.Debug.WriteLine(i.ToString());
    if (i > 50)
        lState.Break();
}
        );
...
```

Ergebnis

Das Ausgabefenster:

Wie Sie dem Ergebnis entnehmen, läuft die Schleife in mehreren Threads, die Reihenfolge der Zahlen scheint allerdings chaotisch zu sein (siehe Bemerkungen).

Bemerkungen

- Der Aufruf von *ParallelLoopState.Break()* informiert die *For*-Operation dass nach der aktuellen Iteration keine weitere mehr ausgeführt werden soll. Jedoch sind alle Iterationen vor der aktuellen noch auszuführen, falls dies noch nicht geschehen ist. Deshalb ist der Aufruf von *Break* vergleichbar mit der Verwendung von *break* innerhalb einer konventionellen *for*-Schleife, aber es ist kein perfekter Ersatz: es gibt beispielsweise keine Garantie dafür, dass nach der aktuellen Iteration keine weitere mehr ausgeführt wird.
- Falls die Ausführung aller Iterationen vor der aktuellen nicht erforderlich ist, sollte *ParallelLoopState.Stop()* den Vorzug vor *Break* erhalten. Wie bei *Break* gibt es aber auch bei *Stop* keine Garantie, ob zukünftige Iterationen ausgeführt werden oder nicht. Wurde eine Schleife vorzeitig beendet, enthält der Rückgabewert von *ParallelLoopstate* relevante Informationen über den Schleifenabschluss.

5.7 Dynamische Programmierung

Die so genannte *dynamische* Programmierung wurde mit C# 4.0 eingeführt. Zu ihren wichtigsten Vorteilen zählen die gravierenden Vereinfachungen bei Verwendung des Reflection-Mechanismus, bei COM-Interop und beim Zugriff auf das HTML DOM. Zur Kompilierzeit erlaubt Ihnen C# jetzt die Ausführung beliebiger Operationen mit Variablen eines dynamischen Typs. Doch ehe wir ins Detail gehen, sollen zunächst einige allgemeine Dinge geklärt werden.

5.7.1 Wozu dynamische Programmierung?

Die meisten C# Entwickler verwenden mehrere Werkzeuge in einer einzigen Applikation, um komplexe Aufgaben zu programmieren. Für WPF Desktop-Anwendungen braucht man beispielsweise C# und XAML. Häufig findet sich Open-Source-Code zum Lösen eines Problems, allerdings kann dieser in einer anderen Sprache vorliegen, z.B. in VB oder F#. Einer der Vorteile von .NET ist die Fähigkeit zur Cross-Language-Interoperabilität, nicht zu Unrecht bezeichnet man ja die Laufzeitumgebung als "Common Language" Runtime (CLR). Microsoft hat zwar dynamische Sprachen wie IronRuby und IronPython entwickelt, für die Entwickler gab es bislang aber keinen einfachen Weg um die Interoperabilität mit dynamischen Sprachen zu gewährleisten. Überzeugen Sie sich nun in diesem Abschnitt davon, wie leicht man mittels dynamischer Programmierung dieses Ziel erreichen kann.

- Wenn Sie mit Hilfe des Reflection-Mechanismus eine Methode auf einem Objekt aufrufen wollen, dann müssen Sie dazu eine Objektreferenz gewinnen, den Typ der verwendeten Bindung ermitteln und dann das Objekt aufrufen. All diese Schritte können mittels dynamischer Methoden vereinfacht werden.
- Bei Silverlight-Anwendungen müssen Sie früher oder später auf das HTML-DOM zugreifen, welches Ihr Silverlight-Control enthält. Auch diese Aufgabe kann die dynamische C#-Programmierung erleichtern.
- Die Realisierung von COM-Interop mittels C# war bislang ziemlich umständlich, besonders weil Sie die exakte Syntax bei Konvertierungen, optionalen Parametern usw. zu beachten hatten[1]. Ein wichtiger Vorteil der dynamischen Programmierung in C# ist die sauberere Syntax in derartig unübersichtlichen Szenarien.

5.7.2 Das Prinzip der dynamischen Programmierung

... lässt sich am besten anhand eines konkreten Beispiels erklären.

Das folgende Beispiel soll das Erzeugen eines Objekts demonstrieren. Drei lose nebeneinander stehende Hilfsklassen dienen als Grundlage.

[1] Einige C#-Entwickler hatten hier neidisch in Richtung VB geschielt, welches bislang eine bessere Unterstützung für COM-Interop bot.

BEISPIEL 5.49: Prinzip der dynamischen Programmierung

C#

Die drei Klassen *CFlugzeug*, *CVogel* und *CStudent* stehen untereinander in keinerlei Beziehung (Vererbung etc.). *CFlugzeug* und *CVogel* verfügen über eine *Fliegen*-Methode, *CStudent* aber nicht.

```
class CFlugzeug
{
    public void Fliegen()
    {
       MessageBox.Show("Das Flugzeug fliegt!");
    }
}

class CVogel
{
    public void Fliegen()
    {
        MessageBox.Show("Der Vogel fliegt!");
    }
}

class CStudent
{
    public void Lernen()
    {
        MessageBox.Show("Der Student lernt!");
    }
}
```

Eine Hilfsmethode, die eine Instanz von *CFlugzeug* liefert:

```
public object getFliegendesObjekt()
{
    return new CFlugzeug();
}
```

Keine der drei Klassen implementiert ein Interface oder eine allgemeine Basisklasse, d h., es existiert keinerlei Code der eine streng typisierte Beziehung zwischen diesen Klassen erzeugt. Später werden Sie sehen, wie man mittels dynamischer Typen trotzdem auf all diese Klassen zugreifen kann.

Ganz bewusst haben wir hier als Rückgabetyp der Methode *getFliegendesObjekt* den Typ *object* gewählt. Wir wollen damit die Bedingungen simulieren die auftreten, wenn man die Referenz auf ein Objekt unbekannten Typs erhält. Das passiert beispielsweise wenn dieses Objekt von der Bibliothek eines Drittanbieters kommt oder von einer Bibliothek stammt, die in einer dynamischen Sprache (wie *IronRuby* oder *IronPython*) geschrieben wurde.

Sie als Programmierer haben die Dokumentation von *getFliegendesObjekt* gelesen und wissen deshalb, dass das Rückgabeobjekt eine *Fliegen*-Methode besitzt, der C#-Compiler aber kann das

wegen der *object*-Referenz nicht herausfinden. Unter dem alten C# mussten Sie deshalb ziemlich umständlich mittels Reflection-Mechanismus die *Fliegen*-Methode auf dem Objekt aufrufen, in C# geht das dank dynamischer Variablen verblüffend einfach.

BEISPIEL 5.50: (Fortsetzung)

C#

Die von *getFlyingObject* gelieferte *object*-Referenz wird der dynamischen Variablen *fliegendesObjekt* zugewiesen.

```
dynamic fliegendesObjekt = getFlliegendesObjekt();
fliegendesObjekt.Fliegen();
```

Der *dynamic* Typ entspricht nur auf den ersten Blick dem *object*-Typ, denn Sie können mit einem *dynamic*-Typ Dinge tun (Methoden aufrufen).

HINWEIS: Beim Kompilieren erlaubt C# Ihnen die Ausführung beliebiger Operationen auf einer dynamischen Variablen.

Das Binden eines dynamischen Typs erfolgt erst zur Laufzeit. Obiges Beispiel ruft *Fliegen* auf der Variablen *fliegendesObjekt* auf. Diese dynamische Variable führt die *Fliegen*-Methode von *CFlugzeug* zur Laufzeit aus.

Falls die Methode, die Sie auf einem dynamischen Objekt aufrufen, nicht existiert, erhalten Sie keine Fehlermeldung des Compilers, sondern einen Laufzeitfehler.

BEISPIEL 5.51: Die Iterator-Methode liefert je eine Instanz von *CFlugzeug, CVogel* und *CStudent*.

C#

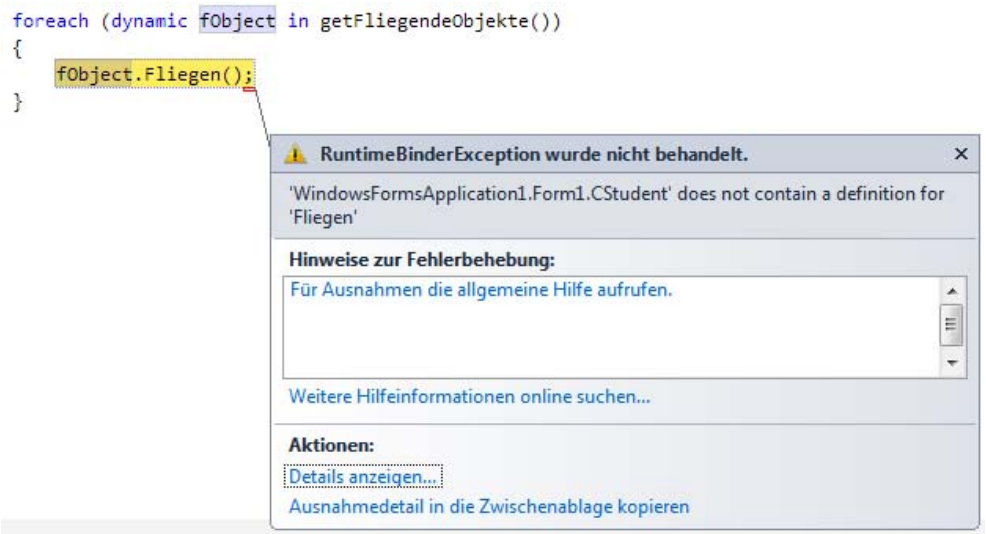
```
public IEnumerable<object> getFliegendeObjekte()
{
    yield return new CFlugzeug();
    yield return new CVogel();
    yield return new CStudent();
}
```

Da *CStudent* keine *Fliegen*-Methode besitzt, generiert der folgende Code einen Laufzeitfehler:

```
foreach (dynamic fO in getFliegendeObjekte())
{
    fOb.Fliegen();
}
```

Der Code iteriert solange anstandslos über *getFliegendeObjekte*, bis eine Instanz geliefert wird (hier die von *CStudent*), die keine *Fliegen*-Methode besitzt. In diesem Fall wird eine Laufzeit-Exception ausgelöst[1].

[1] In diesem Fall ist die Fehlermeldung exakt die gleiche wie Sie sie zur Kompilierzeit sehen würden, wenn die Variable *fliegendesObjekt* nicht dynamisch wäre!



5.7.3 Optionale Parameter sind hilfreich

Die ebenfalls unter C# 4.0 eingeführten optionalen Parameter (siehe 2.7.6) leisten auch bei der dynamischen Programmierung gute Dienste. Dies kann am besten anhand von *Missing.Value* verdeutlicht werden:

Die alten C#-Versionen verlangen die Verwendung von *Missing.Value*, ein statisches Readonly-Feld im *System.Reflection Namespace*, welches als Platzhalter für optionale Werte dient. Da die Rückgabewerte in COM-Interop vom Type *object* sind, mussten Sie bislang eine explizite Konvertierung beim Zuweisen von *ActiveSheet* durchführen.

BEISPIEL 5.52: Ein typischer Excel-Aufruf zum Hinzufügen eines neuen Worksheet

C#

```
var wkBook = excelApp.Workbooks.Add(Missing.Value);
var wkSheetData = (Excel.Worksheet)wkBook.ActiveSheet;
```

Dank dynamischer Programmierung und optionaler Parameter vereinfacht sich obiger Code:

```
dynamic wkBook = excelApp.Workbooks.Add();
Excel.Worksheet wkSheetData = wkBook.ActiveSheet;
```

Wie Sie sehen können Sie, wenn Sie den Typ von *wkBook* explizit auf *dynamic* setzen, die Eigenschaft *ActiveSheet* direkt herauslesen und das Ergebnis dem *Excel.Worksheet* Typ zuweisen.

Das Praxisbeispiel

► 5.9.4 Dynamischer Zugriff auf COM Interop

demonstriert den dynamischen COM Interop-Zugriff auf Microsoft Excel!

5.7.4 Kovarianz und Kontravarianz

Das Füllen eines allgemeineren Typs mit den Werten eines speziellen (tiefer abgeleiteten) Typs bezeichnet man als *Kovarianz*. Im umgekehrten Fall, also wenn Sie einem spezielleren Typ einen allgemeineren Typ zuweisen, spricht man von *Kontravarianz*.

Im vorliegenden Abschnitt werden Sie lernen, wie man Kovarianz und Kontravarianz nicht nur für Arrays, sondern auch für generische Typen implementieren kann.

BEISPIEL 5.53: Einem Objektarray wird ein Stringarray zugewiesen und umgekehrt.

C#
```
string[] strings1 = new string[]  { "Günther", "Erhard", "Michael" };
```
Kovarianz:
```
object[] objects = strings1;
```
Kontravarianz:
```
string[] strings2 = (string[])objects;
```

Wenn man Kovarianz und Kontravarianz in Arrays realisieren kann, so möchte man das verständlicherweise mit der gleichen Semantik auch mit Collections tun. Das folgende Beispiel benutzt die Schnittstelle *IEnumerable* statt eines Arrays, um so Kovarianz und Kontravarianz zu demonstrieren.

BEISPIEL 5.54: Kovarianz und Kontravarianz mit Collections

C#
```
IEnumerable<string> strings3 = strings1;
IEnumerable<object> objects3 = strings3;
IEnumerable<string> strings4 = (IEnumerable<string>)objects3;
```

HINWEIS: In .NET unterstützen alle Collection-Typen Kovarianz und Kontravarianz.

5.8 Weitere Datentypen

Die folgenden Datentypen werden erst ab der Version 4.0 des .NET-Frameworks unterstützt.

5.8.1 BigInteger

Der Typ *System.Numerics.BigInteger* steht für beliebig große ganzzahlige Werte und gestattet das Arbeiten mit einer rein theoretisch unendlich großen Ziffernfolge.

HINWEIS: Fügen Sie, falls noch nicht vorhanden, einen Verweis auf die *System.Numerics.dll* hinzu (Kontextmenü *Verweis hinzufügen...* im Projektexplorer).

Eine neue *BigInteger*-Instanz kann

- mit dem *new*-Schlüsselwort oder
- mit der statischen *Parse()*-Methode erzeugt werden.

BEISPIEL 5.55: Erzeugen eines *BigInteger* aus einem *long*-Wert

C#
```
using System.Numerics;
...
BigInteger bi1 = new BigInteger(1234566m);
```

BEISPIEL 5.56: (Fortsetzung) Erzeugen eines *BigInteger* aus einer Zeichenkette

C#
```
BigInteger bi2 = BigInteger.Parse("12345678901234567890123456789012345678901123456780");
```

Auch die mathematischen Operatoren für Addition, Multiplikation, Subtraktion und Division wurden für diese Klasse überladen.

BEISPIEL 5.57: (Fortsetzung) Multiplikation zweier *BigInteger*

C#
```
BigInteger bi3 = bi2 * bi1;
```

Die folgende Tabelle zeigt die wichtigsten Eigenschaften einer *BigInteger*-Zahl.

Eigenschaft	Beschreibung
Sign	... liefert Vorzeichen des Wertes (-1, 0, 1)
IsEven	... prüft, ob der Wert gerade ist
IsOne	... prüft, ob der Wert Eins ist
IsZero	... prüft, ob der Wert gleich 0 ist
IsPowerOfTwo	... prüft, ob der Wert eine Zweierpotenz ist

BEISPIEL 5.58: Es wird bewiesen, dass die Zahl 2097152 eine Potenz von 2 ist

C#
```
BigInteger bi4 = new BigInteger(2097152);
if (bi4.IsPowerOfTwo)                          // liefert true
        MessageBox.Show(bi4.ToString() + " ist eine Potenz von 2 !");
```

Die folgende Tabelle zeigt einige wichtige statische Methoden der *BigInteger*-Klasse:

Methode	Beschreibung
Add, Subtract, Multiply, Divide	... liefern das Ergebnis der vier Grundrechenarten.
Abs	... ermittelt den Absolutwert einer *BigInteger*-Zahl.

Methode	Beschreibung
DivRem	... dividiert einen *BigInteger*-Wert durch einen anderen und liefert in einem Output-Parameter das Ergebnis und den Rest.
GreatestCommonDivisor	... liefert den größten gemeinsamen Teiler zweier *BigInteger*-Zahlen.
Log, Log10	... ermittelt den natürlichen bzw. dekadischen Logarithmus.
Min, Max	... bestimmt Maximum bzw. Minimum zweier *BigInteger*-Zahlen.
Parse	... konvertiert die Stringdarstellung einer Zahl in ihr *BigInteger*-Äquivalent.
TryParse	... versucht, die Stringdarstellung einer Zahl in ihr *BigInteger*-Äquivalent zu konvertieren. Der Rückgabewert zeigt an, ob die Konvertierung erfolgreich war.
Pow	... berechnet eine bestimmte Potenz für einen BigInteger Wert.
Reminder	... realisiert die Integer-Division von zwei *BigInteger*-Werten und gibt den Rest zurück.

Wie man sieht, können die mathematischen Grundoperationen auch mit entsprechenden Methodenaufrufen realisiert werden.

BEISPIEL 5.59: Alternative Durchführung des Vorgängerbeispiels (Multiplikation zweier *BigInteger*-Zahlen)

C#
```
BigInteger bi3 = BigInteger.Multiply(bi1, bi2);
```

BEISPIEL 5.60: Die ganzzahlige Division *1000000/300000* mit zusätzlicher Ermittlung des Rests

C#
```
BigInteger bi1 = new BigInteger(1000000);
BigInteger bi2 = new BigInteger(300000);
BigInteger bi3 = new BigInteger();

BigInteger bi4 = new BigInteger.DivRem(bi1, bi2, out bi3);
```

bi3 hat jetzt den Wert *100000* und *bi4* den Wert *3*

BEISPIEL 5.61: Die 20.-te Potenz von 10 wird berechnet

C#
```
BigInteger bi1 = new BigInteger(10);
BigInteger bi2 = BigInteger.Pow(bi1, 20);
```

bi2 hat jetzt den Wert von 100000000000000000000

5.8.2 Complex

Der Datentyp *System.Numeric.Complex* repräsentiert eine komplexe Zahl[1]. Auch für diesen Datentyp stehen, wie im folgenden ganz einfachen Beispiel zu sehen ist, die gewöhnlichen mathematischen Operatoren zur Verfügung.

BEISPIEL 5.62: Addition von zwei komplexen Zahlen

C#

Zwei Instanzen von *complex* erzeugen:

```
Complex c1 = new Complex(1.5, -0.3);
Complex c2 = new Complex(-5, 2.1);
```

Operation durchführen:

```
Complex c3 = c1 + c2;
```

Ergebnis auswerten:

- kartesische Koordinaten:

```
double re = c3.Real;        // -3,5
double im = c3.Imaginary;   // 1,8
```

- Polarkoordinaten:

```
double m = c3.Magnitude;    // 3,9357 ...
double p = c3.Phase;        // -0,475 ...
```

Die mathematisch interessierten Leser wird es freuen, dass auch zahlreiche statische Methoden (*Cos*, *Cosh*, *Exp* ...) bereitgestellt werden, mit denen eine Vielzahl weiterer Berechnungen durchführbar ist.

5.8.3 Tuple<>

Die generische *System.Tuple*<>-Klasse ermöglicht es, bis zu acht Elemente unterschiedlichen Typs gemeinsam in einem Tupel typsicher abzuspeichern. Mittels der Eigenschaften *Item1* ... *Item8* kann auf die einzelnen Elemente zugegriffen werden.

BEISPIEL 5.63: Anwendung der *Tuple*-Klasse

C#

Zwei einfache Klassen definieren:

```
public class CLektor
{
    public string Name { get; set; }
}
```

[1] Eine komplexe Zahl besteht immer aus zwei Komponenten, sie kann entweder in kartesischen Koordinaten (Real- und Imaginärteil) oder in Polarkoordinaten (Betrag und Phase) dargestellt werden..

BEISPIEL 5.63: Anwendung der *Tuple*-Klasse

```
public class CAutor
{
   public string Name { get; set; }
}
```

Alle Objekte erzeugen:

```
CLektor le1 = new CLektor {Name = "Sieglinde"};
CAutor au1 = new CAutor {Name = "Walter"};
CAutor au2 = new CAutor {Name = "Thomas"};
```

Jetzt wird ein *Tuple*-Objekt erzeugt:

```
Tuple< CLektor, CAutor, CAutor > tuple1 = new Tuple< CLektor,
                                                     CAutor, CAutor > (le1, au1, au2);
```

Der Zugriff erfolgt über die entsprechenden *Item*-Eigenschaften:

```
CLektor le = tuple1.Item1;
MessageBox.Show(le.Name);                    // zeigt "Sieglinde"
```

oder

```
CAutor au = tuple1.Item3;
MessageBox.Show(au.Name);                    // zeigt "Thomas"
```

HINWEIS: Die Elemente eines Tupels sind schreibgeschützt, können also zur Laufzeit nicht verändert werden.

5.8.4 SortedSet<>

Mit der generischen *System.Collections.Generic.SortedSet<>*-Klasse lassen sich Mengen sortieren. Das Hinzufügen und Einsortieren geschieht mittels *Add()*-Methode.

BEISPIEL 5.64: Erstellen einer Namensliste und Ausgabe in einer *ListBox*

C#

Eine Instanz erzeugen:

```
SortedSet<string> Namen = new SortedSet<string>();
```

Elemente hinzufügen:

```
Namen.Add("Fernando");
Namen.Add("Thomas");
Namen.Add("Max Walter");
```

Ausgabe:

```
foreach (string s in Namen)
    listBox1.Items.Add(s);
```

BEISPIEL 5.64: Erstellen einer Namensliste und Ausgabe in einer *ListBox*

Ergebnis

Die alphabetisch geordnete Ausgabe entspricht dem Standard für Zeichenketten:

```
Fernando
Max Walter
Thomas
```

Dem Konstruktor kann auch ein benutzerdefiniertes Objekt vom generischen Typ *System.Collections.Generic.IComparer<T>* übergeben werden, mit welchem Sie selbst die Sortierfolge bestimmen können.

BEISPIEL 5.65: Die Sortierung der Zeichenketten soll nach der Länge erfolgen

C#

Diese Klasse definiert die neue Sortierung:

```
public class CSortierfolge : IComparer<string>
{
    public int Compare(string a, string b)
    {
        return a.Length - b.Length;
    }
}
```

Die Instanz der sortieren Menge wird nun so erstellt:

```
SortedSet<string> Namen =  new SortedSet<string>(new CSortierfolge());
...
```

Ergebnis

Im Vergleich zum Vorgängerbeispiel sieht die Ausgabe jetzt so aus:

```
Thomas
Fernando
Max Walter
```

5.9 Praxisbeispiele

5.9.1 ArrayList versus generische List

Bei einer *ArrayList* handelt es sich um eine universell einsetzbare (nicht generische) Sammlung (Collection) von Objekten, die vom standardmäßig eingebundenen Namensraum *System.Collections* bereitgestellt wird (siehe Abschnitt 5.3.2).

Generics hingegen sind Klassen bzw. Methoden, denen die Typinformationen erst zur Laufzeit zugewiesen werden. Die variablen Datentypen werden in spitzen Klammern unmittelbar nach dem Klassen- bzw. Methodenbezeichner angegeben. Im Namespace *System.Collections.Generic* finden sich zahlreiche generische Klassen, wie z.B. die *List*, welche als Alternative zur altbekannten *ArrayList* infrage kommt (siehe Abschnitt 5.5.1).

Das vorliegende Beispiel besteht aus zwei getrennten Windows Forms-Anwendungen. In der ersten zeigen wir, wie man in der *ArrayList* sortiert und sucht und wie man deren Inhalt ausliest und in einer *ListBox* zur Anzeige bringt. Die zweite Anwendung löst das gleiche Problem mittels einer generischen *List*.

Oberfläche

Die Oberflächen für beide Anwendungen sind identisch. Auf dem Startformular *Form1* platzieren Sie eine *ListBox*, eine *TextBox* und drei *Button*s (siehe Laufzeitabbildung am Schluss).

Quellcode (mit ArrayList)

```
...
using System.Collections;

public partial class Form1 : Form
{
    ...
    private ArrayList  al = null;
```

Nach Programmstart wird die *ArrayList* erzeugt, mit Werten gefüllt und angezeigt:

```
    private void Form1_Load(object sender, EventArgs e)
    {
        al = new ArrayList(); al.Add("Das"); al.Add("ist");  al.Add("ein");  al.Add("Test");
        al.Add("mit"); al.Add("der"); al.Add("ArrayList"); al.Add("Das"); al.Add("ist");
        al.Add("ok.");
        listeAnzeigen(al);
    }
```

Die Routine zur Anzeige des Inhalts in der *ListBox*:

```
    private void listeAnzeigen(ArrayList al)
    {
        listBox1.Items.Clear();
        for (int i = 0; i < al.Count; i++)
        {
            listBox1.Items.Add(al[i]);
        }
    }
```

Aufsteigend sortieren:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        al.Sort();
       listeAnzeigen(al);
    }
```

Absteigend sortieren:

```
    private void button2_Click(object sender, System.EventArgs e)
    {
        al.Reverse();
        listeAnzeigen(al);
    }
```

Suchen:

```
    private int ix = -1;          // Index des letztmaligen Vorkommens

    private void button3_Click(object sender, System.EventArgs e)
    {
        char lf = (char) 10;      // für Zeilenvorschub
        int i = al.IndexOf(textBox1.Text, ix+1);     // Suche wird ab ix fortgesetzt
        if (i < 0)
        {
            MessageBox.Show("Das Element wurde nicht gefunden!" + lf +
                                                    " Suche beginnt wieder von vorn!");
            ix = -1;
            listBox1.SelectedIndex = -1;       // Zeilenmarkierung entfernen
        }
        else
        {
            listBox1.SelectedIndex = i;        // gefundenes Wort markieren
            ix = i;
        }
    }
}
```

Quellcode (mit generischer List)

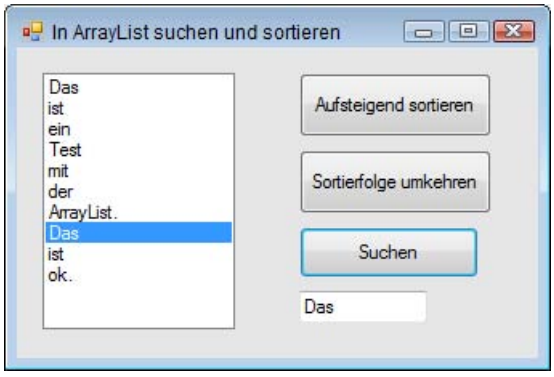
Alle Änderungen gegenüber dem *ArrayList*-Beispiel sind fett hervorgehoben. Wie Sie sehen, können *ArrayList* und *List* direkt ausgetauscht werden, da sie über dieselben Methoden (*Add*, *Sort*, *Reverse*, *IndexOf*) verfügen.

```
using System.Collections.Generic;

public partial class Form1 : Form
{
    private List<string> gl = null;
    ...
    private void Form1_Load(object sender, EventArgs e)
    {
        gl = new List<string>();
        gl.Add("Das");
        gl.Add("ist");
        ...
```

```
            listeAnzeigen(al);
        }

        private void listeAnzeigen(List<string> gl)
        {
            listBox1.Items.Clear();
            for (int i = 0; i < gl.Count; i++)
            {
                listBox1.Items.Add(gl[i]);
            }
        }
        ...
        // restlicher Code siehe ArrayList-Version
}
```

Test

Beide Programme zeigen dasselbe Verhalten. Die Abbildung zeigt eine erfolgreiche Suche ohne vorherige Sortierung:



Ansichten der sortierten Listen:

Bemerkungen

Gegenüber der *ArrayList* bietet die generische *List* alle Vorzüge einer typsicheren Programmierung:

- Das Boxing von Wertetypen ist nicht mehr erforderlich, weil *List<string>* in unserem Beispiel jetzt "weiß", welcher Datentyp zu speichern ist und wie viel Speicher man dafür reservieren muss.
- Statt eines Laufzeitfehlers erhalten Sie bereits einen Compiler-Fehler, falls Sie einen falschen Datentyp in der List speichern wollen.
- Eine Typumwandlung (Typecasting) entfällt, wenn Sie auf die Werte der generischen List zugreifen wollen.

5.9.2 Generische IEnumerable-Interfaces implementieren

Das .NET-Framework stellt zwei Arten von Standardschnittstellen für die Enumeration und den Vergleich von Collections zur Verfügung:

- Interfaces für die traditionellen, nicht typsicheren Collections (siehe Abschnitt 5.3) und
- Interfaces für die neueren generischen Collections (siehe Abschnitt 5.5).

In unserem Beispiel speichert eine Klasse *CTestListe* ein String-Array und implementiert dazu die generische Schnittstelle *IEnumerable<string>*. Dies ist erforderlich, um beispielsweise mittels *foreach*-Anweisung über diese selbst definierte Collection iterieren zu können.

Doch halt, schießen wir mit einer extra Klasse *CTestListe* nicht mit Kanonen auf Spatzen?

Wenn es lediglich darum geht, mit *foreach* über die Stringliste zu iterieren, so genügen wenige Codezeilen.

BEISPIEL 5.66: Auslesen einer Stringliste und Anzeige in einer *ListBox*

C#
```
IEnumerable<string> liste = new string[] { "Günter","Erhard", "Michael"};

foreach (string s in liste)
{
    listBox1.Items.Add(s);
}
```

Wir aber wollen noch etwas mehr, als es obiger Code zeigt. Für mehr Komfort, wie das Hinzufügen von Einträgen, indizierter Zugriff etc., lohnt sich durchaus das Anlegen einer eigenen Klasse.

Oberfläche

Platzieren Sie auf dem Startformular einer neuen Windows Forms-Anwendung eine *ListBox* und einen *Button*.

Quellcode CTestListe

Fügen Sie zum Projekt eine Klasse *CTestListe* hinzu:

```
using System.Collections.Generic;
using System.Text;

class CTestListe : IEnumerable<string>    // generische Schnittstelle implementieren
{
    private string[] strings;     // Stringarray speichert den Inhalt der Testliste
    private int pos = 0;          // Positionszähler
```

Im Konstruktor wird die Liste mit den übergebenen Strings initialisiert:

```
   public CTestListe(params string[] initStrings )
   {
      strings = new String[6];              // genügend Platz für die Strings reservieren
      foreach ( string s in initStrings )   // die übergebene Stringliste wird eingelesen:
          strings[pos++] = s;
   }
```

Die Implementierung des Enumerators ist Pflicht:

```
   IEnumerator<string> IEnumerable<string>.GetEnumerator()
   {
       foreach ( string s in strings )      // Listeninhalt wird durchlaufen
           yield return s;                 // Enumerator wird geliefert
   }
```

Eine Methode, die einen String an das Ende der Liste anhängt:

```
   public void Add( string s )
   {
       strings[pos] = s;
       pos++;
   }
```

Schließlich gönnen wir uns noch einen **Indexer**, der einen indizierten Zugriff auf die Liste wie bei einem Array erlaubt:

```
   public string this[int ix]
   {
       get
       {
           if ( ix < 0 || ix >= strings.Length )
           {  }
           return strings[ix];
       }
       set
       {
           strings[ix] = value;
       }
   }
```

HINWEIS: Leider würde obiger Code einen Compilerfehler verursachen, da die gleichnamige (nichtgenerische!) *GetEnumerator()*-Methode nicht implementiert wurde. Weil deren Signatur identisch ist, muss sie explizit implementiert werden.

Für den folgenden kleinen Workaround ist der Namespace *System.Collections* zusätzlich einzubinden:

```
      IEnumerator IEnumerable.GetEnumerator()
      {  throw new NotImplementedException();  }
   }
}
```

Quellcode Form1

```
public partial class Form1 : Form
{
    private void button1_Click(object sender, EventArgs e)
    {
```

Eine neue Testliste erzeugen und mit drei Einträgen initialisieren:

```
        CTestListe liste = new CTestListe( "Günter", "Erhard", "Michael" );
```

Weitere Einträge hinzufügen und indizierten Zugriff testen:

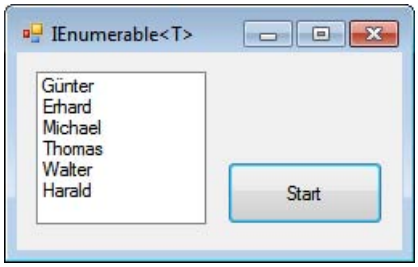
```
        liste.Add( "Thomas" );
        liste.Add( "Walter" );
        liste[5] = "Harald";
```

Alle Einträge lesen:

```
        foreach ( string s in liste )
           listBox1,Items.Add(s);
     }
}
```

Test

Das Programm zeigt die erwarteten Ergebnisse.



Bemerkungen

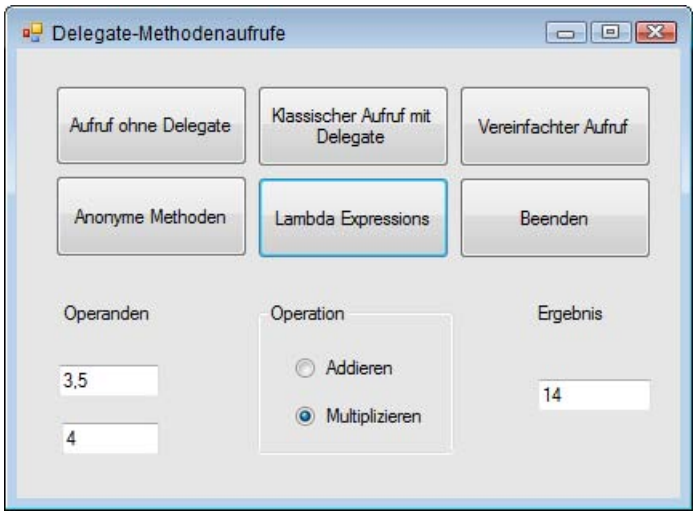
- Klassen, die die *IEnumerable*-Schnittstelle implementieren, können einen Enumerator zurückgeben.
- *IEnumerable<T>* hat nur eine einzige Methode, *GetEnumerator( )*, deren Aufgabe es ist, eine Implementatierung von *IEnumerator<T>* zu liefern.
- Die *foreach*-Schleife benutzt automatisch das *IEnumerable<T>* Interface, wobei *GetEnumerator( )* aufgerufen wird.
- C# unterstützt mit dem Schlüsselwort *yield* das Erzeugen eines Enumerators.

5.9.3 Delegates, anonyme Methoden, Lambda Expressions

Dieses Beispiel will dem Einsteiger den Weg zu den etwas schwierigeren Sprachelementen von C# ebnen, an denen aber kein Weg vorbei führt, will man sich in die höheren Regionen der Programmentwicklung begeben. Grundlage bildet die Addition und Multiplikation von Fließkommazahlen. Dieses Problem wird auf fünf verschiedene und voneinander unabhängige Arten realisiert, wodurch sich gute Vergleichsmöglichkeiten ergeben, um das Wesen von Delegaten (Methodenzeiger), anonymen Methoden und Lambda Expressions zu erfassen.

Oberfläche

Ein Formular mit sechs *Button*s, zwei in einer *GroupBox* befindlichen *RadioButton*s und drei *TextBox*en bildet unsere Testoberfläche (siehe folgende Laufzeitabbildung).



Über jede der Schaltflächen (außer der zum Beenden) werden die Rechenoperationen auf unterschiedliche Weise ausgeführt.

Quellcode

Zunächst deklarieren wir den Delegate-Typ *opDeleg*, welcher der Signatur der beiden Methoden *add()* und *mult()* entspricht, die wir danach deklarieren:

```
public delegate double opDeleg(float x, float y);

public partial class Form1 : Form
{
    ...

    private double add(float x, float y)
    {
        return x + y;
    }

    private double mult(float x, float y)
    {
        return x * y;
    }
```

HINWEIS: Obige zwei Methoden werden nur für die ersten drei Varianten benötigt.

Gewissermaßen als "Aufwärmübung" werden über den Button "Aufruf ohne Delegate" die Rechenoperationen zunächst auf herkömmliche Weise (also ohne Verwendung von Delegates) ausgeführt.

```
    private void button1_Click(object sender, EventArgs e)
    {
        float a = Convert.ToSingle(textBox1.Text);
        float b = Convert.ToSingle(textBox2.Text);
        double res = 0;
        if (radioButton1.Checked) res = add(a, b);
        else res = mult(a, b);
        textBox3.Text = res.ToString();
    }
```

Nun wollen wir zeigen, wie dieselben Operationen unter Verwendung von Delegate-Objekten realisiert werden können:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Der erste Schritt ist das Deklarieren einer Variablen vom Typ des Delegaten (das Prinzip entspricht dem Deklarieren einer Objektvariablen für eine Klasse):

```
        opDeleg myDlg;
```

In Abhängigkeit von der eingestellten Rechenoperation wird das Delegate-Objekt entweder mit der *add()*- oder mit der *mult()*-Methode erzeugt (entspricht dem Instanziieren einer Klasse):

```
        if (radioButton1.Checked) myDlg = new opDeleg(add);
        else myDlg = new opDeleg(mult);
        float a = Convert.ToSingle(textBox1.Text);
        float b = Convert.ToSingle(textBox2.Text);
```

Aufruf des Delegate-Objekts und Ergebnisanzeige:

```
        double res = myDlg(a,b);
        textBox3.Text = res.ToString();
    }
```

Der Schreibaufwand für den Code ist keinesfalls geringer als bei der ersten Variante. Eine gewisse Vereinfachung, die allerdings erst auf den zweiten Blick ersichtlich ist, bringt die dritte Variante, sie verwendet eine vereinfachte Instanziierung des Delegaten (ohne *new*):

```
    private void button3_Click(object sender, EventArgs e)
    {
        opDeleg myDlg;
        if (radioButton1.Checked) myDlg = add;
        else myDlg = mult;
        float a = Convert.ToSingle(textBox1.Text);
        float b = Convert.ToSingle(textBox2.Text);
        double res = myDlg(a, b);
        textBox3.Text = res.ToString();
    }
```

Alle bisherigen Varianten erfordern den Aufruf der (benannten) Methoden *add()* und *mult(),* wie sie zu Beginn implementiert wurden. Ganz ohne diese Methoden kommt die folgende Variante aus, die so genannte *anonyme Methoden* verwendet:

```
    private void button4_Click(object sender, EventArgs e)
    {
```

Die folgende Instanziierung der Delegate-Objekte *addDlg* und *multDlg* erfolgt mittels der in den geschweiften Klammern implementierten anonymen (namenlosen) Methoden:

```
        opDeleg addDlg = delegate(float x, float y)
        {
            return x + y;
        };

        opDeleg multDlg = delegate(float x, float y)
        {
            return x * y;
        };
```

Wie man sieht, handelt es sich bei anonymen Methoden um keine Methoden im herkömmlichen Sinn, sondern nur um das "Anhängen" von Implementierungscode, was eine kompaktere und übersichtlichere Programmierung ermöglicht.

Die Anwendung beider Delegate-Objekte zeigt keine Besonderheiten:

```
        float a = Convert.ToSingle(textBox1.Text);
        float b = Convert.ToSingle(textBox2.Text);
        double res = 0;

        if (radioButton1.Checked) res = addDlg(a, b);
        else res = multDlg(a,b);
        textBox3.Text = res.ToString();
    }
```

Hat man das Prinzip der anonymen Methoden verstanden, so ist es nur noch ein kleiner Schritt bis hin zu den ab .NET 3.0 neu eingeführten so genannten *Lambda Expression*s, wie der folgende Code für die fünfte und letzte Variante zeigt:

```
    private void button5_Click(object sender, EventArgs e)
    {
```

Die folgenden beiden Codezeilen instanziieren die Delegaten *addDlg* und *multDlg*, den Platz der anonymen Methoden in der Vorgängervariante nehmen diesmal aber die Lambda Expressions ein. Letztere sind durch einen Doppelpfeil gekennzeichnet, auf dessen linker Seite in Klammern die an die Methode zu übergebenden Argumente stehen, auf der rechten Seite findet sich die Implementierung der Methode (ist bei mehreren Anweisungen in geschweifte Klammern einzuschließen):

```
        opDeleg addDlg = (float x, float y) => x + y;
        opDeleg multDlg = (float x, float y) => x * y;
```

Der restliche Code entspricht der Vorgängervariante:

```
        float a = Convert.ToSingle(textBox1.Text);
        float b = Convert.ToSingle(textBox2.Text);
        double res = 0;

        if (radioButton1.Checked) res = addDlg(a, b);
        else res = multDlg(a, b);
        textBox3.Text = res.ToString();
    }
    ...
}
```

Test

Geben Sie in die beiden linken Textfelder irgendwelche Zahlen ein (bei der standardmäßigen deutschen Kultureinstellung ist das Dezimaltrennzeichen das Komma und das Tausender-Trennzeichen der Punkt). Wählen Sie anschließend die auszuführende Operation (Addition/Multiplikation) und starten Sie die Ausführung über eine beliebige Schaltfläche.

Jede der fünf Varianten muss zum exakt gleichen Ergebnis führen!

5.9.4 Dynamischer Zugriff auf COM Interop

Die Office-Programmierung ist ein beliebter Tummelplatz für dynamische Variablen, wie sie seit C# 4.0 möglich sind. In diesem Beispiel soll die Excel-Automation ohne und mit dynamischer Programmierung gegenübergestellt werden.

Oberfläche

Ein Formular *Form1* und zwei *Buttons* genügen.

Quellcode (klassische Programmierung)

Der Aufruf von *Microsoft Excel* ohne dynamische Programmierung.

```
using System.Reflection;
using Excel = Microsoft.Office.Interop.Excel;

private static void executeOldInterOp()
{
            var eApp = new Excel.Application();

            eApp.Visible = true;

            var wkBook = eApp.Workbooks.Add(Missing.Value);
            var wsData = (Excel.Worksheet)wkBook.ActiveSheet;

            wsData.Cells[1, 1] = "Name";
            wsData.Cells[1, 2] = "Email";

            var data = new string[,]
            {
                {"Walter Doberenz", "doberenz@doko-buch.de"},
                {"Thomas Gewinnus", "thomas@gewinnus.de"}
            };

            var dRange = wsData.get_Range("A2", "B3");
            dRange.Value2 = data;

            var wkSheetEnd =(Excel.Worksheet)wkBook.Worksheets.Add(
                         Missing.Value, wsData, Missing.Value, Missing.Value);

            wkSheetEnd.Cells[2, 2] = "Testeintrag";
            wkBook.SaveAs( "test.xls", Missing.Value, Missing.Value,
                        Missing.Value, Missing.Value, Missing.Value,
                        Excel.XlSaveAsAccessMode.xlShared,
                        Missing.Value, Missing.Value,
                        Missing.Value, Missing.Value, Missing.Value);
    }
```

Der Aufruf:

```
private void button1_Click(object sender, EventArgs e)
{
    executeOldInterOp();
}
```

Quellcode (dynamische Programmierung)

Der Aufruf von *Microsoft Excel* mit dynamischer Programmierung.

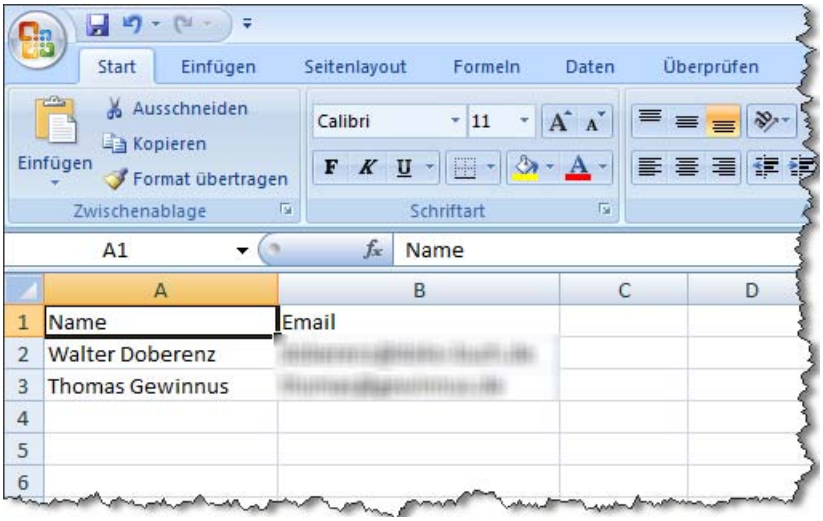
```
private static void executeNewInterOp()
{
    var eApp = new Excel.Application();
    eApp.Visible = true;

    dynamic wkBook = eApp.Workbooks.Add();
    Excel.Worksheet wsData = wkBook.ActiveSheet;

    wsData.Cells[1, 1] = "Name";
    wsData.Cells[1, 2] = "Email";
    var data = new string[,]
    {
            {"Walter Doberenz", "autor1@buch.de"},
            {"Thomas Gewinnus", "autor2@buch.de"}
    };
    var dRange = wsData.get_Range("A2", "B3");
    dRange.Value2 = data;
    Excel.Worksheet wsEnd = wkBook.Worksheets.Add(After: wsData);

    wsEnd.Cells[2, 2] = "Testeintrag";
    wkBook.SaveAs("test.xls",  Excel.XlSaveAsAccessMode.xlShared);
}
```

Der Aufruf:

```
private void button2_Click(object sender, EventArgs e)
{
      executeNewInterOp();
}
```

Test

Nach dem Start können Sie die beiden Varianten ausprobieren (siehe folgende Abbildung).



Bemerkungen

- Vergleichen Sie beide Beispiele um sich davon zu überzeugen, dass der dynamische Code einfacher und transparenter ist.
- Beim Entwickeln von Anwendungen die auf COM-Interop zugreifen, braucht man sich keine Sorgen um das Verteilen der Interop-Assembly zu machen. Seit Visual Studio 2010 ist es möglich, die Typen aus der Interop-Assembly direkt in Ihre Projekt- Assembly einzubetten. Öffnen Sie dazu den Knoten *Verweise* im Projektmappen-Explorer, klicken Sie mit der rechten Maustaste auf die Interop-Assembly und ändern Sie die Eigenschaft *Interop-Typen einbetten* in *True*.

Kapitel 6

Einführung in LINQ

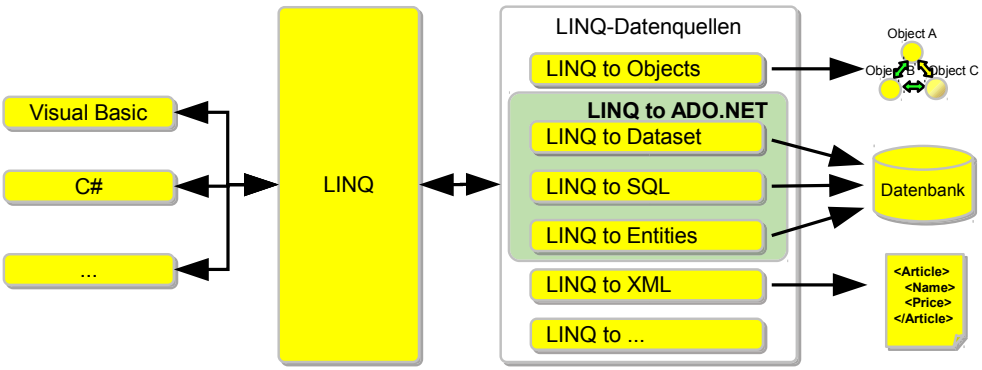
LINQ (Language Integrated Query) ist ein relativ neues C#-Sprachfeature (C# 3.0). Bei *LINQ to Objects* handelt es sich um die allgemeinste und grundlegendste LINQ-Implementierung, welche auch die wichtigsten Bausteine für die übrigen LINQ-Implementierungen liefert. In einer SQL-ähnlichen Syntax können miteinander verknüpfte Collections/Auflistungen abgefragt werden, die die *IEnumerable*-Schnittstelle implementieren.

6.1 LINQ-Grundlagen

Der wichtigste Vorteil von LINQ ist die standardisierte Möglichkeit, nicht nur Tabellen in einer relationalen Datenbank, sondern auch Textdateien, XML-Dateien und andere Datenquellen in einer SQL-ähnlichen Syntax abzufragen. Ein zweiter Vorteil ist die Fähigkeit, diese standardisierten Methoden von jeder .NET-konformen Sprache (wie zum Beispiel C# oder VB) aus aufrufen zu können.

6.1.1 Die LINQ-Architektur

Die folgende Abbildung soll die grundsätzliche Architektur von LINQ verdeutlichen.



Je nach Standort des Betrachters besteht LINQ einerseits aus einer Menge von Werkzeugen zur Arbeit mit Daten, was in den verschiedenen LINQ-Implementationen (LINQ to Objects, LINQ to DataSets, LINQ to SQL, LINQ to Entities und LINQ to XML) zum Ausdruck kommt. Andererseits besteht LINQ aus einer Menge von Spracherweiterungen.

LINQ-Implementierungen

LINQ eröffnet zahlreiche Varianten für den Zugriff auf verschiedenste Arten von Daten. Diese sind in den verschiedenen LINQ-Implementationen (auch als "LINQ Flavors" d h. "Geschmacksrichtungen" bezeichnet) enthalten. Folgende LINQ-Provider wurden bereits als Bestandteile des .NET-Frameworks bereitgestellt (siehe obige Abbildung):

- LINQ to Objects (arbeitet mit Collections die *IEnumerable* implementieren),
- LINQ to XML (Zugriff auf XML Strukturen),
- LINQ to SQL (Zugriff auf SQL Datenbanken),
- LINQ to DataSet (arbeitet auf Basis von DataSets) und
- LINQ to Entities (verwendet das ADO.NET Entity Framework als ORM).

Diese LINQ-Provider/Implementationen bilden eine Familie von Tools, die einzeln für bestimmte Aufgaben eingesetzt oder aber auch für leistungsfähige Lösungen mit einem Mix aus Objekten, XML und relationalen Daten kombiniert werden können.

HINWEIS: Wir werden uns in diesem Kapitel hauptsächlich auf **LINQ to Objects** beschränken, da dieserProvider die grundlegende Technologie bereitstellt und die übrigen Flavours mehr eine Angelegenheit der Datenbankliteratur sind (siehe unser Buch [Datenbankprogrammierung mit Visual C# 2012]).

Nochmals sei hier betont, dass LINQ eine offene Technologie ist, der jederzeit neue Provider hinzugefügt werden können. Die im .NET Framework enthaltenen Implementationen bilden lediglich eine Basis, die eine Menge von Grundbausteinen (Abfrage-Operatoren, Abfrageausdrücke, Abfragebäume) bereitstellt.

Die Einführung von LINQ machte mehrere Ergänzungen der .NET-Programmiersprachen erforderlich, von denen einige bereits in diesem Kapitel (Lambda-Ausdrücke) bzw. im einführenden Sprachkapitel 2 (Typinferenz) bzw. im OOP-Kapitel 3 (Objektinitialisierer) besprochen wurden. Auf zwei weitere Sprachfeatures (anonyme Typen und Erweiterungsmethoden), ohne die LINQ nicht möglich wäre, wollen wir im Folgenden eingehen.

6.1.2 Anonyme Typen

Darunter verstehen wir einfache namenlose Klassen, die vom Compiler automatisch erzeugt werden und die nur über Eigenschaften und dazugehörige private Felder verfügen. "Namenlos" bedeutet, dass uns der Name der Klasse nicht bekannt ist und man deshalb keinen direkten Zugriff auf

die Klasse hat. Lediglich eine Instanz steht zur Verfügung, die man ausschließlich lokal, d h. im Bereich der Deklaration, verwenden kann.

Die Deklaration anonymer Typen erfolgt mittels *Typinferenz* (Schlüsselwort *var*, siehe 2.2.10) und eines anonymen Objektinitialisierers, d h., man lässt beim Initialisieren einfach den Klassennamen weg. Der Compiler erzeugt die anonyme Klasse anhand der Eigenschaften im Objekt-Initialisierer und anhand des jeweiligen Typs der zugewiesenen Werte.

BEISPIEL 6.1: Eine Objektvariable *person* wird aus einer anonymen Klasse instanziiert.

C#
```
var person = new { Vorname = "Maxhelm", Nachname = "Müller", Alter = 53};
```

Der Compiler generiert hierfür intern (in MSIL-Code) die folgende Klasse:

```
internal class ??????
{
    private string _vorname;
    private string _nachname;
    private int _alter;

    public string Vorname
    {
        get { return _vorname; }
        set { _vorname = value; }
    }

    public string Nachname
    {
        get { return _nachname; }
        set { _nachname = value; }
    }

    public int Alter
    {
        get { return _alter; }
        set { _alter = value; }
    }
}
```

HINWEIS: Da der Typ der Eigenschaften aus der jeweiligen Klasse bzw. Struktur des im Objekt-Initialisierer zugewiesenen Wertes abgeleitet wird, darf man hier nicht *null* zuweisen, denn der Compiler kann in diesem Fall den Datentyp der Eigenschaft nicht bestimmen.

BEISPIEL 6.2: Fehlerhafter Code, der Compiler kann den Typ der Eigenschaft *Alter* nicht ableiten.

C#
```
var person = new { Voname = "Maxhelm" , Nachname = "Müller", Alter = null};   // Fehler!!!
```

Sobald eine weitere anonyme Klasse deklariert wird, bei der im Objektinitialisierer Eigenschaften mit gleichem Namen, Typ und in der gleichen Reihenfolge wie bei einer anderen bereits vorhandenen anonymen Klassen angegeben sind, verwendet der Compiler die gleiche anonyme Klasse und es sind untereinander Zuweisungen möglich.

BEISPIEL 6.3: Da Name, Typ und Reihenfolge der Eigenschaften im Objektinitialisierer bei *person* (siehe oben) und *kunde* identisch sind, ist eine direkte Zuweisung möglich.

C#
```
var kunde = new { Vorname = "Siegbast", Nachname = "Krause", Alter = 29};
person = kunde;
```

6.1.3 Erweiterungsmethoden

Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Bereits C# 3.0 führte aber eine neue Syntax ein, die das direkte Hinzufügen neuer Methoden zu einer vorhandenen Klasse erlaubt. Diese so genannten *Erweiterungsmethoden* werden als statische Methoden in einer neuen statischen Klasse implementiert und können dann wie eine normale Methode (d.h. Instanzenmethode) des erweiterten Datentyps aufgerufen werden. Um eine Methode als Erweiterungsmethode zu deklarieren, wird vor dem ersten Parameter das Schlüsselwort *this* angegeben. Der Argumenttyp des ersten Parameters bezieht sich auf die zu erweiternde Klasse bzw. Struktur. Wird die Erweiterungsmethode dann aufgerufen, übergibt der Compiler die Instanz des erweiterten Typs als erstes Argument an die Methode.

BEISPIEL 6.4: Zwei Erweiterungsmethoden (*mult* und *abs*) für die Basisklasse *System.Int32*

C#
```
public static class IntExtension
{
    public static int mult(this int instanz, int faktor)       // 1. Erweiterungsmethode
    {
        return instanz * faktor;
    }

    public static int abs(this int instanz)                   // 2. Erweiterungsmethode
    {
       if (instanz < 0)
           return -1 * instanz;
       return instanz;
    }
}
```

Der Test:

```
int zahl = -95;
textBox1.Text = zahl.mult(7).ToString();      //  -665
textBox2.Text = zahl.abs().ToString();        //    95
```

In diesem Beispiel kann man nun die Erweiterungsmethoden *mult* und *abs* für jede Integer Variable so nutzen, als wären diese Methoden direkt in der Basisklasse *System.Int32* als Instanzenmethoden implementiert.

HINWEIS: Falls in *System.Int32* bereits eine *abs*-Methode mit der gleichen Signatur wie die gleichnamige Erweiterungsmethode existieren würde, so hätte die in *System.Int32* bereits vorhandene Methode Vorrang vor der Erweiterungsmethode.

6.2 Abfragen mit LINQ to Objects

LINQ stellt bekanntlich grundlegende Abfragefunktionen in einer SQL-ähnlichen Syntax bereit. Dazu gehören zunächst als wichtigster Standard das Angeben einer Quelle (*from*), das Festlegen der zurückzugebenden Daten (*select),* das Filtern (*where*) und das Sortieren (*orderby*). Hinzu kommt eine Fülle weiterer Operatoren, wie z.B. für das Gruppieren, Verknüpfen und Sammeln von Datensätzen, auf welche wir aber erst später eingehen wollen.

6.2.1 Grundlegendes zur LINQ-Syntax

Die LINQ-Abfrageoperatoren sind als Erweiterungsmethoden (siehe Abschnitt 6.1.3) definiert und in der Regel auf beliebige Objekte, die *IEnumerable<T>* implementieren, anwendbar.

BEISPIEL 6.5: Gegeben sei die Auflistung

C#
```
string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
                    "Juli", "August", "September", "Oktober", "November", "Dezember" };
```

Die folgende LINQ-Abfrage selektiert die Monatsnamen mit einer Länge von 6 Buchstaben, wandelt sie in Großbuchstaben um und ordnet sie alphabetisch.

```
var expr = from s in monate
           where s.Length == 6
           orderby s
           select s.ToUpper();
```

Die Ergebnisanzeige:

```
foreach (string item in expr) listBox1.Items.Add(item);
```

Das Resultat in einem Listenfeld:

```
AUGUST
JANUAR
```

Obiges Beispiel demonstriert das allgemeine Format einer LINQ-Abfrage:

SYNTAX: `from ... < where ... orderby ... > select ...`

Eine LINQ-Abfrage muss immer mit *from* beginnen. Im Wesentlichen durchläuft *from* eine Liste von Daten. Dazu wird eine Variable benötigt, die jedem einzelnen Datenelement in der Quelle entspricht.

HINWEIS: Wer die Sprache SQL kennt, der wird zunächst darüber irritiert sein, warum eine LINQ-Abfrage mit *from* und nicht mit *select* beginnt. Der Grund hierfür ist der, dass nur so ein effektives Arbeiten mit der IntelliSense von Visual Studio möglich ist. Da zuerst die Datenquelle ausgewählt wird, kann die IntelliSense geeignete Typmitglieder für die Objekte der Auflistung anbieten.

Weiterhin erkennen Sie, wie vom neuen Sprachfeature der lokalen Typinferenz (implizite Variablendeklaration) Gebrauch gemacht wird, denn die Anweisung

```
var expr = from s in monate ...
```

ist für den Compiler identisch mit

```
IEnumerable<string> expr = from s in monate ...
```

6.2.2 Zwei alternative Schreibweisen von LINQ Abfragen

Grundsätzlich sind für LINQ Abfragen zwei gleichberechtigte Schreibweisen möglich:

- Query Expression-Syntax[1] (Abfrage-Syntax)
- Extension Method-Syntax (Erweiterungsmethoden-Syntax)

Bis jetzt haben wir aber nur die Query Expression-Syntax verwendet. Um die volle Leistungsfähigkeit von LINQ auszuschöpfen, sollten Sie aber beide Syntaxformen verstehen.

BEISPIEL 6.6: Die LINQ-Abfrage des obigen Beispiels in Extension Method-Syntax.

C#

```
var expr = monate
           .Where(s => s.Length == 6)
           .OrderBy(s => s)
           .Select(s => s.ToUpper());
```

Oder kompakt in einer Zeile:

```
var expr = monate.Where(s => s.Length == 6).OrderBy(s => s).Select(s => s.ToUpper());
```

Wie Sie sehen, verwenden wir bei dieser Notation Erweiterungsmethoden und Lambda-Ausdrücke. Aber auch eine Kombination von *Query Expression-Syntax* mit *Extension Method-Syntax* ist möglich.

BEISPIEL 6.7: Obiges Beispiel in gemischter Syntax

C#

```
var expr = (from s in monate where s.Length == 6  select s.ToUpper()).OrderBy(s => s);
```

[1] Die *Extension Method -Syntax* wird auch als *Dot Notation-Syntax* bezeichnet.

Hier wurde ein Abfrageausdruck in runde Klammern eingeschlossen, gefolgt von der Erweiterungsmethode *OrderBy*. Solange wie der Abfrageausdruck ein *IEnumerable* zurückgibt, kann darauf eine ganze Kette von Erweiterungsmethoden folgen.

Die Query Expression-Syntax (Abfragesyntax) ermöglicht das Schreiben von Abfragen in einer SQL-ähnlichen Weise. Wo immer es möglich ist, empfehlen wir, vor allem der besseren Lesbarkeit wegen, die Verwendung dieser Syntax. Letztendlich konvertiert jedoch der Compiler alle Queries in die andere, auf Erweiterungsmethoden basierende, Syntaxform. Dabei wird z.B. die Filterbedingung *where* einfach in den Aufruf einer Erweiterungsmethode namens *Where* der *Enumerable*-Klasse übersetzt, die im Namespace *System.Linq* definiert ist.

Allerdings unterstützt die Query Expression-Syntax nicht jeden standardmäßigen Abfrage-Operator bzw. kann nicht jeden unterstützen den Sie selbst hinzufügen. In einem solchen Fall sollten Sie direkt die Extension Method-Syntax verwenden.

Abfrageausdrücke unterstützen eine Anzahl verschiedener "Klauseln", z. B. *where*, *select*, *orderby*, *groupby* und *join*. Wie bereits erwähnt, lassen sich diese Klauseln in die gleichwertigen Operator-Aufrufe übersetzen, die wiederum über Erweiterungsmethoden implementiert werden. Die enge Beziehung zwischen den Abfrageklauseln und den Erweiterungsmethoden, welche die Operatoren implementieren, erleichtert ihre Kombination, falls die Abfragesyntax keine direkte Klausel für einen erforderlichen Operator unterstützt.

6.2.3 Übersicht der wichtigsten Abfrage-Operatoren

Die Klasse *Enumerable* im Namespace *System.Linq* stellt zahlreiche Abfrageoperatoren für LINQ to Objects bereit und definiert diese als Erweiterungsmethoden für Typen die *IEnumerable<T>* implementieren.

HINWEIS: Kommen bei der Extension Method-Syntax (Erweiterungsmethoden-Syntax) Abfrage-Operatoren bzw. -Methoden zur Anwendung, so sollten wir bei der Query Expression-Syntax (Abfrage-Syntax) präziser von Abfrage-Klauseln bzw. -Statements sprechen.

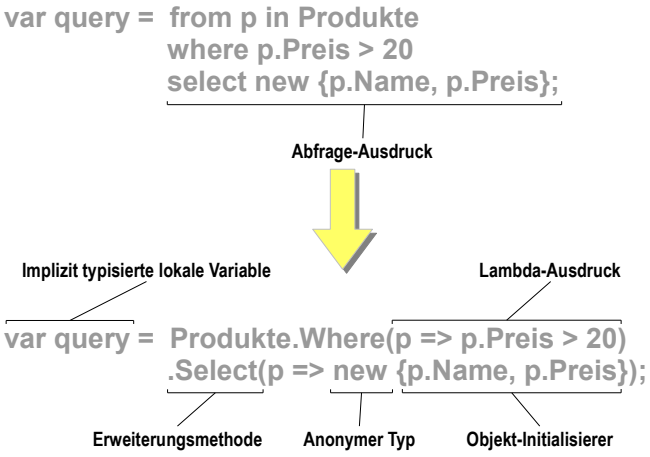
Die folgende Tabelle zeigt die wichtigsten standardmäßigen Abfrage-Operatoren von LINQ.

Bezeichnung der Gruppe	Operator
Beschränkungsoperatoren (Restriction)	*Where*
Projektionsoperatoren (Projection)	*Select, SelectMany*
Sortieroperatoren (Ordering)	*OrderBy, ThenBy*
Gruppierungsoperatoren (Grouping)	*GroupBy*
Quantifizierungsoperatoren (Quantifiers)	*Any, All, Contains*
Aufteilungsoperatoren (Partitioning)	*Take, Skip, TakeWhile, SkipWhile*
Mengenoperatoren (Sets)	*Distinct, Union, Intersect, Except*

Bezeichnung der Gruppe	Operator
Elementoperatoren (Elements)	*First, FirstOrDefault, ElementAt*
Aggregatoperatoren (Aggregation)	*Count, Sum, Min, Max, Average*
Konvertierungsoperatoren (Conversion)	*ToArray, ToList, ToDictionary*
Typumwandlungsoperatoren (Casting)	*OfType<T>*

Die folgende Abbildung illustriert an einem Beispiel, wie einige der bereits im Vorgängerabschnitt diskutierten neuen Sprach-Features in LINQ-Konstrukten zur Anwendung kommen und wie die Abfrage-Syntax vom Compiler in die äquivalente Erweiterungsmethoden-Syntax umgesetzt wird.

Vergleich zwischen Abfrage-Syntax (oben) und Erweiterungsmethoden-Syntax (unten):



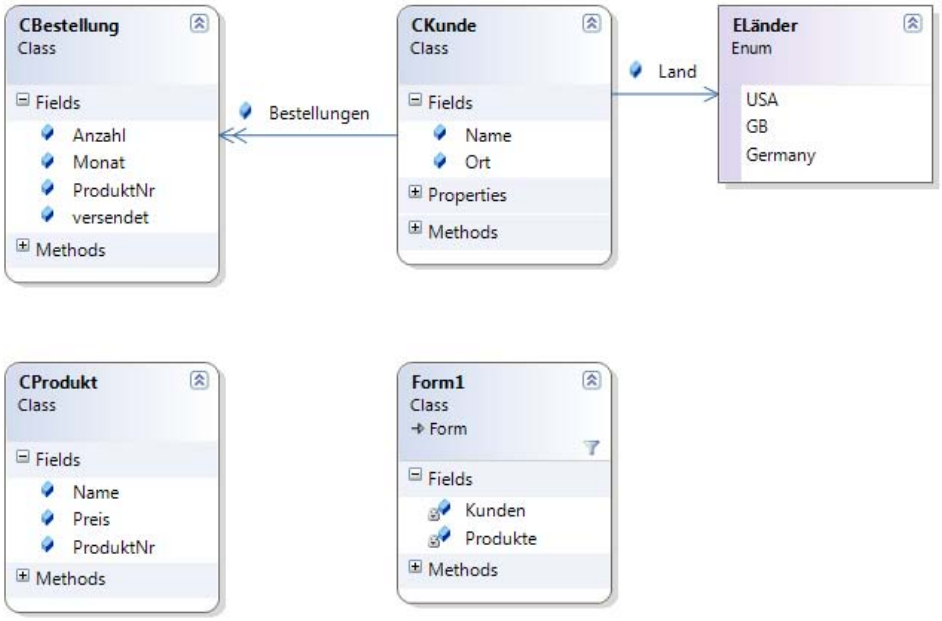
6.3 LINQ-Abfragen im Detail

Das Ziel der folgenden Beispiele ist nicht die vollständige Erläuterung aller in obiger Tabelle aufgeführten Operatoren und deren Überladungen, sondern vielmehr eine Demonstration des prinzipiellen Aufbaus von Anweisungen zur Abfrage von Objektauflistungen.

In der Regel werden beide Syntaxformen (Query Expression-Syntax und Extension Method-Syntax) gegenübergestellt, denn nur so erschließt sich am ehesten das allgemeine Verständnis für die auch für den SQL-Kundigen nicht immer leicht durchschaubare Logik der LINQ-Operatoren bzw. -Abfragen.

Für die Beispiele zu LINQ to Objects wird überwiegend auf eine Datenmenge zugegriffen, deren Struktur das folgende, mit dem Klassendesigner von Visual Studio entwickelte, Diagramm zeigt.

HINWEIS: Die verwendeten Daten haben ihren Ursprung nicht in einer Datenbank, sondern werden per Code erzeugt (Listing siehe Beispieldaten zum Buch).



6.3.1 Die Projektionsoperatoren Select und SelectMany

Diese Operatoren "projizieren" die Inhalte einer Quell-Auflistung in eine Ziel-Auflistung, die das Abfrageergebnis repräsentiert.

Select

Der Operator macht die Abfrageergebnisse über ein Objekt verfügbar, welches *IEnumerable<T>* implementiert.

BEISPIEL 6.8: Komplette Produktliste ausgegeben (Extension Method-Syntax):

C#

```
var allProdukte = Produkte.Select(p => p.Name);
```

Alternativ die Query Expression-Syntax:

```
var allProdukte = from p in Produkte select p.Name;
```

Die Ausgabe der Ergebnisliste:

```
foreach (var p in allProdukte) listBox1.Items.Add(p);
```

Ergebnis

Der Inhalt des Listenfeldes sollte dann etwa folgenden Anblick bieten:

```
Marmelade
Quark
Mohrrüben
...
```

BEISPIEL 6.9: Das Abfrageergebnis wird auf einen anonymen Typ projiziert, der als Tupel definiert ist.

C#

```
var expr = Kunden.Select(k => new {k.Name, k.Ort});
foreach (var k in expr) listBox1.Items.Add(k);
```

Ergebnis

Das Ergebnis:

```
{Name = Walter, Ort = Altenburg}
{Name = Thomas, Ort = Berlin}
...
```

SelectMany

Stände nur der *Select*-Operator zur Verfügung, so hätte man zum Beispiel bei der Abfrage der Bestellungen für alle Kunden eines bestimmten Landes das Problem, dass das Ergebnis vom Typ *IEnumerable<CBestellung>* wäre, wobei es sich bei jedem Element um ein Array mit den Bestellungen eines einzelnen Kunden handeln würde. Um einen praktikableren, d h. weniger tief geschachtelten, Ergebnistyp zu erhalten, wurde der Operator *SelectMany* eingeführt.

BEISPIEL 6.10: Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

C#

```
var bestellungen = Kunden
                .Where(k => k.Land == ELänder.Germany)
                .SelectMany(k => k.Bestellungen);
```

Alternativ der Abfrageausdruck in Query Expression Syntax:

```
var bestellungen =
                from k in Kunden
                where k.Land == ELänder.Germany
                from b in k.Bestellungen
                select b;
```

Das Auslesen des Ergebnisses der Abfrage:

```
            foreach (var b in bestellungen) listBox1.Items.Add(b);
```

Ergebnis

Die Ausgabe (Voraussetzung ist entsprechende Überschreibung der *ToString()*-Methode der Klasse *CBestellung*):

```
ProdNr: 2 , Anzahl: 4 , Monat: März, Versand: False
ProdNr: 1 , Anzahl: 11, Monat: Juni , Versand: True
...
```

6.3.2 Der Restriktionsoperator Where

Dieser Operator schränkt die Ergebnismenge anhand einer Bedingung ein. Sein prinzipieller Einsatz wurde bereits in den Vorgängerbeispielen hinreichend demonstriert. Außerdem können auch Indexparameter verwendet werden, um die Filterung auf bestimmte Indexpositionen zu begrenzen.

BEISPIEL 6.11: Die Kunden an den Positionen 2 und 3 der Kundenliste sollen angezeigt werden.

C#

```
var expr =  Kunden
            .Where((k, index) => ((index >= 2) && (index < 4)))
            .Select(k => k.Name);
foreach (var kd in expr) listBox1.Items.Add(kd);
```

Ergebnis

Die Ausgabe:

```
Holger
Fernando
```

6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy

Diese Operatoren bewirken ein Sortieren der Elemente innerhalb der Ergebnismenge.

OrderBy/OrderByDescending

Dieses nette Pärchen ermöglicht Sortieren in auf- bzw. absteigender Reihenfolge.

BEISPIEL 6.12: Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (der teuerste zuerst).

C#

```
var prod = Produkte
          .Where(p => p.Preis <= 20)
          .OrderByDescending(p => p.Preis)
          .Select(p => new { p.Name, p.Preis });
```

Oder alternativ als Abfrageausdruck:

```
var prod = from p in Produkte
           where p.Preis <= 20
           orderby p.Preis descending
           select new { p.Name, p.Preis };
```

Die Ausgabeschleife:

```
foreach (var p in prod) listBox1.Items.Add(p);
```

Ergebnis

```
{Name = Käse, Preis = 20}
{Name = Mohrrüben, Preis = 15}
...
```

ThenBy/ThenByDescending

Diese Operatoren verwendet man, wenn nacheinander nach mehreren Schlüsseln sortiert werden soll. Da *ThenBy* und *ThenByDescending* nicht auf den Typ *IEnumerable<T>*, sondern nur auf den Typ *IOrderedSequence<T>* anwendbar sind, können diese Operatoren nur im Anschluss an *OrderBy*/*OrderByDescending* eingesetzt werden.

BEISPIEL 6.13: Alle Kunden sollen zunächst nach ihrem Land und dann nach ihren Namen sortiert werden.

C#

```
var knd = Kunden
          .OrderBy(k => k.Land)
          .ThenBy(k => k.Name)
          .Select(k => new { k.Land, k.Name});
```

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
          orderby k.Land, k.Name
          select new { k.Land, k.Name };
```

Die Ausgabe ...

```
          foreach (var ku in knd) listBox1.Items.Add(ku);
```

Ergebnis

... führt in beiden Fällen zu einem Ergebnis wie diesem:

```
{Land = USA, Name = Fernando}
{Land = USA, Name = Holger}
{Land = GB, Name = Alice}
{Land = Germany, Name = Thomas}
{Land = Germany, Name = Walter}
```

Reverse

Dieser Operator bietet eine einfache Möglichkeit, um die Aufeinanderfolge der Elemente im Abfrageergebnis umzukehren.

BEISPIEL 6.14: Das Vorgängerbeispiel mit umgekehrter Reihenfolge der Ergebniselemente

C#

```
var knd = Kunden.OrderBy(k => k.Land).ThenBy(k => k.Name).Select(k => new { k.Land, k.Name })
          .Reverse();
```

Ergebnis

```
{Land = Germany, Name = Walter}
{Land = Germany, Name = Thomas}
...
```

6.3.4 Der Gruppierungsoperator GroupBy

Dieser Operator kommt dann zum Einsatz, wenn das Abfrageergebnis in gruppierter Form zur Verfügung stehen soll. *GroupBy* wählt die gewünschten Schlüssel-Elemente-Zuordnungen aus der abzufragenden Auflistung aus.

BEISPIEL 6.15: Alle Kunden nach Ländern gruppieren

C#

```
var knd = Kunden
          .GroupBy(k => k.Land);
```

BEISPIEL 6.15: Alle Kunden nach Ländern gruppieren

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
          group k by k.Land;
```

Durchlaufen der Ergebnismenge:

```
            foreach (IGrouping<ELänder, CKunde> kdGroup in knd)
            {
                listBox1.Items.Add(kdGroup.Key);
                foreach (var kd in kdGroup)
                    listBox1.Items.Add("  " + kd);
            }
```

Ergebnis

Der Gruppenschlüssel (*kdGroup.Key*) ist hier das Land. Die Standardausgabe der Gruppenelemente erfolgt entsprechend der überschriebenen *ToString()*-Methode der Klasse *CKunden* (siehe Beispieldaten zum Buch):

```
Germany
   Walter - Altenburg - Germany
   Thomas - Berlin - Germany
USA
   Holger - Washington - USA
   Fernando - New York - USA
GB
   Alice - London - GB
```

Der *GroupBy*-Operator existiert in mehreren Überladungen, die alle den Typ *IEnumerable<IGrouping<K, T>* liefern. Die generische Schnittstelle *IGrouping<K, T>* definiert einen spezifischen Schlüssel vom Typ *K* für die Gruppenelemente (Typ *T*) .

Der Typ der äußeren Schleifenvariablen *IGrouping<ELänder, CKunde>* kann auch implizit deklariert werden, sodass sich der Schleifenkopf im obigen Beispiel wie folgt vereinfachen lässt:

```
            foreach(var kdGroup in knd)
```

Wenn nicht der standardmäßige, sondern ein benutzerdefinierter Elemente-Selektor zum Einsatz kommen soll, muss eine weitere Überladung von *GroupBy* verwendet werden (siehe folgendes Beispiel).

BEISPIEL 6.16: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst, als Gruppenelemente werden allerdings nur die Namen der Kunden ausgegeben.

C#

```
var knd = Kunden
          .GroupBy(k => k.Land, k => k.Name);
          foreach (var kdGroup in knd)
          {
              listBox1.Items.Add(kdGroup.Key);
              foreach (var kd in kdGroup)
```

BEISPIEL 6.16: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst, als Gruppenelemente werden allerdings nur die Namen der Kunden ausgegeben.

```
                    listBox1.Items.Add("  " + kd);
            }
```

Ergebnis

```
Germany
   Walter
   Thomas
USA
   Holger
   Fernando
GB
   Alice
```

BEISPIEL 6.17: Alle Produkte werden nach ihren Anfangsbuchstaben gruppiert.

C#

```
var prodGroups = Produkte
                 .GroupBy(p => p.Name[0], p => p.Name);
                 foreach (var pGroup in prodGroups)  // var ersetzt IGrouping<char, string>
                 {
                     listBox1.Items.Add(pGroup.Key);
                     foreach (var p in pGroup)
                         listBox1.Items.Add("  " + p);
                 }
```

Ergebnis

```
M
  Marmelade
  Mohrrüben
  Mehl
Q
  Quark
K
  Käse
H
  Honig
```

C#

Zum gleichen Ergebnis führt der folgende Code unter Verwendung eines Abfrageausdrucks:

```
var prodGroups = from p in Produkte
                 group p by p.Name[0] into g
                 select new { firstLetter = g.Key, prods = g };
                 foreach (var g in prodGroups)
                 {
                    listBox1.Items.Add(g.firstLetter);
                    foreach (var p in g.prods)
                        listBox1.Items.Add(p.Name);
                 }
```

6.3.5 Verknüpfen mit Join

Mit diesem Operator definieren Sie Beziehungen zwischen verschiedenen Auflistungen. Im folgenden Beispiel werden Bestelldaten auf Produkte projiziert.

BEISPIEL 6.18: Die Bestellungen aller Kunden werden aufgelistet.

C#

```
var bestprod = Kunden.SelectMany(k => k.Bestellungen)
                     .Join(Produkte, b => b.ProduktNr, p => p.ProduktNr,
                      (b, p) => new { b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet });
```

Alternativ die Notation in Abfragesyntax:

```
var bestprod = from k in Kunden
                          from b in k.Bestellungen
                          join p in Produkte on b.ProduktNr equals p.ProduktNr
                          select new { b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet };
```

Beim Vergleich (*equals*) ist zu beachten, dass zuerst der Schlüssel der äußeren Auflistung (*b.ProduktNr*) und dann der der inneren Auflistung (*p.ProduktNr*) angegeben werden muss.

Die Anzeigeroutine:

```
foreach (var bp in bestprod)
{
   listBox1.Items.Add(bp);
}
```

Ergebnis

Das Ergebnis liefert die Übersicht über alle Bestellungen:

```
{Monat = März, ProduktNr = 2, Name = Quark, Preis = 10, versendet = False}
{Monat = Juni, ProduktNr = 1, Name = Marmelade, Preis = 5, versendet = False}
{Monat = November, ProduktNr = 3, Name = Mohrrüben, Preis = 15, versendet = True}
{Monat = November, ProduktNr = 5, Name = Honig, Preis = 25, versendet = True}
{Monat = Juni, ProduktNr = 6, Name = Mehl, Preis = 30, versendet = False}
{Monat = Februar, ProduktNr = 4, Name = Käse, Preis = 20, versendet = True}
...
```

6.3.6 Aggregat-Operatoren

Zum Abschluss unserer Stippvisite bei den LINQ-Operatoren wollen wir noch einen kurzen Blick auf eine weitere wichtige Familie werfen. Diese Operatoren, zu denen *Count*, *Sum*, *Max*, *Min*, *Average* etc. gehören, setzen Sie ein, wenn Sie verschiedenste Berechnungen mit den Elementen der Datenquelle durchführen wollen.

Count

Die von diesem Operator durchzuführende Aufgabe ist sehr einfach, es wird die Anzahl der Elemente in der abzufragenden Auflistung ermittelt.

BEISPIEL 6.19: Alle Kunden sollen, zusammen mit der Anzahl der von ihnen aufgegebenen Bestellungen, angezeigt werden.

C#

```
var kdn = Kunden.Select(k => new { k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count() });
```

Oder das Gleiche in Abfrage-Syntax:

```
var kdn = from k in Kunden
          select new { k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count() };
```

Wir iterieren durch die Ergebnismenge:

```
          foreach (var ku in kdn)
               listBox1.Items.Add(ku);
```

Ergebnis

```
{Name = Walter, Ort = Altenburg, AnzahlBest = 1}
{Name = Thomas, Ort = Berlin, AnzahlBest = 2}
...
```

Wie Sie sehen, scheint die Anwendung dieser Operatoren einfach und leicht verständlich zu sein.

Sum

Wie es der Name schon vermuten lässt, können mit diesem Operator verschiedenste Summen aus den Elementen der Quell-Auflistung gebildet werden. Zunächst ein einfaches Beispiel.

BEISPIEL 6.20: Die Summe aller Preise der Produktliste

C#

```
var total = Produkte.Sum(p => p.Preis);
```

Die alternative Abfrage-Syntax (eigentlich gemischte Syntax):

```
var total = (from p in Produkte select p.Preis).Sum();
```

Die Ausgabe

```
listBox1.Items.Add(total);
```

... liefert mit den ursprünglichen Beispieldaten den Wert *105*.

Das folgende Beispiel ist nicht mehr ganz so trivial, da sich hier der *Sum*-Operator innerhalb einer verschachtelten Abfrage versteckt.

BEISPIEL 6.21: Die Gesamtsumme aller aufgegebenen Bestellungen wird ermittelt.

C#

```
var expr = from k in Kunden
           join b in ( from k in Kunden from b in k.Bestellungen join p in Produkte
                on b.ProduktNr equals p.ProduktNr
                select new { k.Name, BestellBetrag = b.Anzahl * p.Preis }
             ) on k.Name equals b.Name into KundenMitBest
             select new { k.Name, TotalBetrag = KundenMitBest.Sum(b => b.BestellBetrag)};
    foreach (var k in expr)
           listBox1.Items.Add(k);
```

BEISPIEL 6.21: Die Gesamtsumme aller aufgegebenen Bestellungen wird ermittelt.

Ergebnis

```
{Name = Walter, TotalBetrag = 40}
{Name = Thomas, TotalBetrag = 340}
...
```

6.3.7 Verzögertes Ausführen von LINQ-Abfragen

Normalerweise werden LINQ-Ausdrücke nicht bereits bei ihrer Definition, sondern erst bei Verwendung der Ergebnismenge ausgeführt. Damit hat man die Möglichkeit, nachträglich Elemente zu der abzufragenden Auflistung hinzuzufügen bzw. zu ändern, ohne dazu die Abfrage nochmals neu erstellen zu müssen.

BEISPIEL 6.22: Alle Produkte, die mit dem Buchstaben "M" beginnen, sollen ermittelt werden.

C#

```
var MProds = Produkte.Where(p => p.Name[0] == 'M').Select(p => p.Name);
```

Oder alternativ in Abfragesyntax:

```
var MProds = from p in Produkte where p.Name[0] =='M' select p.Name;
```

Die Ergebnismenge wird das erste Mal durchlaufen und angezeigt:

```
foreach (var prod in MProds)
     listBox1.Items.Add(prod);
listBox1.Items.Add("----------");
```

Anschließend ändern wir ein Element in der der Abfrage zugrundeliegenden Quelle

```
Produkte[0].Name = "Milch";
```

... und durchlaufen die Ergebnismenge ein zweites Mal:

```
foreach (var prod in MProds)
     listBox1.Items.Add(prod);
```

Ergebnis

Die Ausgabe im Listenfeld zeigt, dass in der zweiten Ergebnismenge das geänderte Element erscheint:

```
Marmelade
Mohrrüben
Mehl
----------
Milch
Mohrrüben
Mehl
```

Wir sehen, dass die definierte Abfrage immer dann ausgeführt wird, wenn wir (wie hier in der *foreach*-Schleife) auf das Abfrageergebnis (*MProds*) zugreifen.

Abfragen dieser Art bezeichnet man deshalb auch als "verzögerte Abfragen"[1]. Mitunter aber ist dieses Verhalten nicht erwünscht, d h., man möchte das Abfrageergebnis nicht verzögert, sondern sofort nach Definition der Abfrage zur Verfügung haben. Das hätte auch den Vorteil, dass sich die Performance verbessert, weil die Abfrage nicht (wie im Beispiel innerhalb der *foreach*-Schleife) immer wieder zur Ausführung kommt. Abhilfe schafft hier die im nächsten Abschnitt beschriebene Anwendung von Konvertierungsmethoden.

6.3.8 Konvertierungsmethoden

Zu dieser Gruppe gehören *ToArray*, *ToList*, *ToDictionary*, *AsEnumerable*, *Cast* und *ToLookup*. Sowohl die Methoden *ToArray* als auch *ToList* forcieren ein sofortiges Durchführen der Abfrage.

BEISPIEL 6.23: Das Vorgängerbeispiel wird wiederholt, diesmal aber wird das Abfrageergebnis in einem Array zwischengespeichert.

C#

```
var MProds = (Produkte.Where(p => p.Name[0] == 'M').Select(p => p.Name)).ToArray();
```

bzw.:

```
var MProds = (from p in Produkte where p.Name[0] == 'M' select p.Name).ToArray();
...
```

Ergebnis

Die Änderung der Quellfolge bleibt ohne Konsequenz für das Abfrageergebnis:

```
Produkte[0].Name = "Milch";
...
```

Ausgabe:

```
Marmelade
Mohrrüben
Mehl
-----------
Marmelade
Mohrrüben
Mehl
```

6.3.9 Abfragen mit PLINQ

PLINQ ist eine parallele Implementierung von LINQ to Objects und kombiniert die Einfachheit und Lesbarkeit der LINQ Syntax mit der Leistungsfähigkeit der parallelen Programmierung. PLINQ besitzt das komplette Angebot an Standard-Abfrageoperatoren und hat zusätzliche Operatoren für parallele Operationen.

Als Reaktion auf die zunehmende Verfügbarkeit von Mehrprozessorplattformen bietet PLINQ eine einfache Möglichkeit, die Vorteile paralleler Hardware einschließlich herkömmlicher Mehrprozessorcomputer und der neueren Generation von Mehrkernprozessoren zu nutzen.

[1] *deferred query execution*

HINWEIS: In vielen Szenarien kann PLINQ signifikant die Geschwindigkeit von LINQ to Objects-Abfragen steigern, da es alle verfügbaren Prozessoren des Computers nutzt.

Wer bereits mit LINQ vertraut ist, dem wird der Umstieg auf PLINQ kaum Sorgen bereiten. Die Verwendung von PLINQ entspricht meistens exakt der von LINQ-to-Objects und LINQ-to-XML. Sie können beliebige der bereits bekannten Operatoren nutzen, wie zum Beispiel *Join*, *Select*, *Where* usw.

Damit können Sie auch unter PLINQ Ihre bereits vorhandenen LINQ-Abfragen auf gewohnte Weise weiter verwenden, wenn Sie dabei einen wesentlichen Unterschied beachten:

HINWEIS: Parallelisieren Sie die Abfrage durch Aufruf der Erweiterungsmethode *AsParallel*!

Die Erweiterungsmethode *AsParallel* gehört zur *System.Linq.ParallelQuery*-Klasse, diese ist in der *System.Core.dll* enthalten und repräsentiert eine parallele Sequenz. *AsParallel* kann auf jeder Datenmenge ausgeführt werden, die *IEnumerable<T>* implementiert.

Der Aufruf von *AsParallel* veranlasst den C#-Compiler, die parallele Version der Standard-Abfrageoperatoren zu binden. Damit übernimmt PLINQ die weitere Verarbeitung der Abfrage.

BEISPIEL 6.24: Eine einfache LINQ-Abfrage über eine Liste von Integer-Zahlen

C#

```
int[] zahlen = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Oder auch:

```
IEnumerable<int> zahlen = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var q = from x in zahlen
        where x > 3
        orderby x descending
        select x;
```

Erst beim Iterieren über die Liste wird die Abfrage ausgeführt:

```
foreach (var z in q) listBox1.Items.Add(z.ToString());    // 9, 8, 7, 6, 5, 4
```

Um dieselbe Abfrage mittels PLINQ auszuführen, ist lediglich *AsParallel* auf den Daten aufzurufen:

BEISPIEL 6.25: Die Abfrage im obigen Beispiel mit PLINQ

C#

```
...
var q = from x in zahlen.AsParallel()
        where x > 3
        orderby x descending
        select x;
...
```

Die Abfragen in obigen Beispielen wurden in Query Expression-Syntax geschrieben. Alternativ kann man natürlich auch die Extension Method-Syntax[1] verwenden.

BEISPIEL 6.26: Beide obigen Abfragen in Erweiterungsmethoden-Syntax

C#

Einfache LINQ-Version:

```
...
var q = zahlen
        .Where(x => x > 3)
        .OrderByDescending(x => x)
        .Select(x => x);
...
```

PLINQ-Version:

```
...
 var q = zahlen.AsParallel()
        .Where(x => x > 3)
        .OrderByDescending(x => x)
        .Select(x => x);
...
```

Nach dem Aufruf der *AsParallel*-Methode führt PLINQ transparent die Erweiterungsmethoden (*Where*, *OrderBy*, *Select*,...) auf allem verfügbaren Prozessoren aus. Genauso wie LINQ realisiert auch PLINQ eine verzögerte Ausführung von Abfragen, d.h., erst beim Durchlaufen der *foreach*-Schleife, beim Direktaufruf von *GetEnumerator*, oder beim Eintragen der Ergebnisse in eine Liste (*ToList*, *ToDictionary*,...) wird die Datenmenge abgefragt. Dann kümmert sich PLINQ darum, dass bestimmte Teile der Abfrage auf verschiedenen Prozessoren laufen, was mit versteckten multiplen Threads umgesetzt wird. Sie als Programmierer brauchen das nicht zu verstehen, Sie merken lediglich an der höheren Performance, dass die Prozessoren besser ausgelastet werden.

Probleme mit der Sortierfolge

Wie sollte es anders sein, bei genauerem Hinsehen werden Sie feststellen, dass es doch nicht ganz so unkompliziert ist, LINQ-Abfragen zu parallelisieren. Ganz abgesehen davon, dass die Parallelisierung nicht immer den erhofften Geschwindigkeitszuwachs bringt, habe wir es noch mit einem schwierigen und vor allem nicht gleich erkennbaren Problem zu tun: der Sortierfolge. Diese bereitet im Zusammenhang mit der parallelen Verarbeitung teilweise recht große Probleme, da auch bei einer geordneten Ausgangsmenge nicht eindeutig ist, in welcher Reihenfolge die Elemente durch PLINQ verarbeitet werden. Je nach LINQ-Operator kann es zu recht merkwürdigen Ergebnissen kommen[2].

[1] Der Compiler konvertiert die Query Expression-Syntax in die Extension Method-Syntax, sodass letztendlich bei beiden Syntaxformen Erweiterungsmethoden aufgerufen werden.

[2] Dies ist auch von der Anzahl der Prozessoren und der Größe der Datenmenge abhängig.

Aus diesem Grund wurde die Erweiterungsmethode *AsOrdered* eingeführt. Verwenden Sie diese im Zusammenhang mit *AsParallel,* wird die Sortierfolge der Ausgangsmenge in jedem Fall beibehalten.

BEISPIEL 6.27: Verwendung von *AsOrdered*

C#

```
int[] zahlen = new int[] { 7, 4, 2, 3, 1, 6, 11, 5, 10, 8, 9, 13, 12 };
var q = (from x in zahlen.AsParallel().AsOrdered()
where x > 3
select x).Take(5);
```

Das Ergebnis wird in jedem Fall

```
7, 4, 6, 11, 5
```

sein. Lassen Sie *AsOrdered* weg, sind weder die obige Reihenfolge noch die Zahlen eindeutig bestimmbar. Unter Umständen kann auch

```
13, 7, 11, 6, 5
```

ausgegeben werden.

Wie sich Sortierfolgen auf bestimmte Operatoren auswirken, beschreibt im Detail die folgende Webseite:

LINK: `http://msdn.microsoft.com/de-de/library/dd460677(v=vs.110).aspx`

HINWEIS: Grundsätzlich jedoch gilt: Vermeiden Sie im Zusammenhang mit PLINQ die Anwendung von Sortieroperationen, diese machen die Vorteile von PLINQ durch erhöhten Verwaltungsaufwand meist wieder zunichte.

6.4 Praxisbeispiele

6.4.1 Die Syntax von LINQ-Abfragen verstehen

In diesem Praxisbeispiel lernen Sie den prinzipiellen Aufbau von LINQ-Abfragen kennen. Im Zusammenhang damit kommen die ab .NET 3.0 eingeführten Sprachfeatures wie Typinferenz, Lambda-Ausdrücke und Erweiterungsmethoden zum Einsatz.

Prinzipiell gibt es zwei verschiedene Syntaxformen für LINQ-Abfragen:

- *Query Expression Syntax*
  Hier werden Standard-Query-Operatoren verwendet.
- *Extension Method Syntax*
  Hier kommen Erweiterungsmethoden zum Einsatz.

Im Folgenden werden wir beide Syntaxformen demonstrieren, um den Inhalt eines Integer-Arrays zu verarbeiten. Außerdem wird eine Mischform vorgeführt.

Oberfläche

Auf dem Startformular *Form1* finden eine *ListBox* und vier *Button*s ihren Platz (siehe Laufzeitansicht).

Quellcode

```
...
using System.Linq;
...
public partial class Form1 : Form
{
```

Das abzufragende Integer-Array enthält irgendwelche ganzzahligen Werte:

```
    private int[] zahlen = {5, -4, 18, 26, 0, 19, 16, 2, -1, 0, 9, -5, 8, 15, 19 };
```

Die Abfrage in Query Expression Syntax:

```
    private void button1_Click(object sender, EventArgs e)
    {
```

Im Abfrageergebnis sollen alle Zahlen, die größer als 10 sind, enthalten sein und nach ihrer Größe sortiert werden. Im Abfrageausdruck kommen die SQL-ähnlichen Standard-Abfrageoperatoren (*from*, *where*, *orderby*, *select*) zum Einsatz:

```
        var expr = from z in zahlen
                           where z > 10
                           orderby z
                           select z;
```

Die Anzeige:

```
        foreach (int z in expr)  listBox1.Items.Add(z.ToString());
    }
```

Dieselbe Abfrage in Extension Method Syntax:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Hier werden im Abfrageausdruck so genannte Erweiterungsmethoden (*Where*, *OrderBy*, *Select*) zusammen mit Lambda-Ausdrücken (... => ...) benutzt:

```
        var expr = zahlen
                          .Where(z => z > 10)
                          .OrderBy(z => z)
                          .Select(z => z);
```

Die Anzeige:

```
       foreach (int z in expr)  listBox1.Items.Add(z.ToString());
    }
```

Die gleiche Abfrage in gemischter Syntax:

```
    private void button3_Click(object sender, EventArgs e)
    {
```

Hier wird der erste (geklammerte) Teil des Abfrageausdrucks in Query Expression Syntax, und der zweite (mit einem Punkt eingeleitete) Teil in Extension Method Syntax geschrieben:

```
        var expr = (from z in zahlen
                                where z > 10
                                select z).
                                OrderBy(z => z);
```

Die Anzeige:

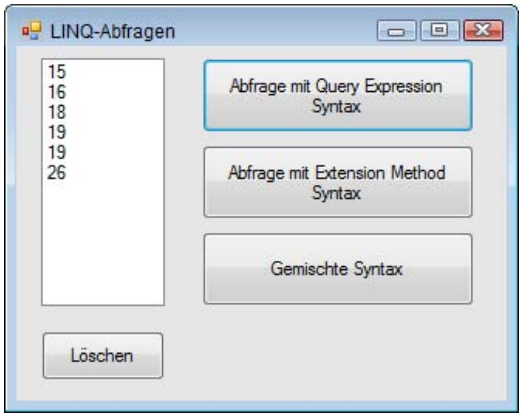
```
        foreach (int z in expr)  listBox1.Items.Add(z.ToString());
    }
```

ListBox-Inhalt löschen:

```
    private void button4_Click(object sender, EventArgs e)
    {
        listBox1.Items.Clear();
    }
}
```

Test

Egal, welche der drei Schaltflächen Sie klicken, das Ergebnis wird stets dasselbe sein:



Bemerkungen

- Unter *Typinferenz* versteht man ein Sprachmerkmal welches es erlaubt, dass der Datentyp **lokaler** Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass explizit der Typ angegeben werden muss (stattdessen wird das Schlüsselwort *var* vorangestellt). Wie wir gesehen haben, erweist sich dieses Feature vor allem für *anonyme Typen* als praktisch bzw. notwendig.

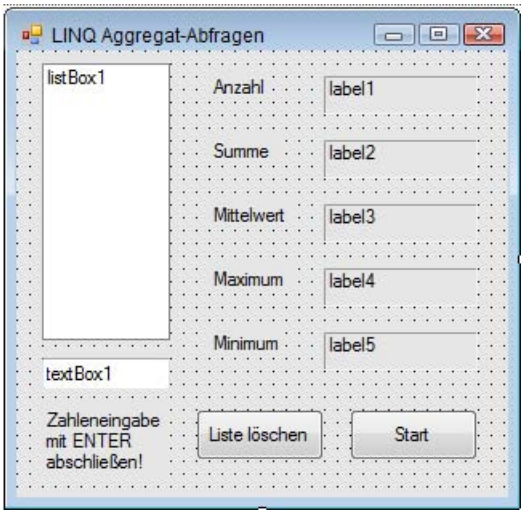
- Bei den *Lambda-Ausdrücken* handelt es sich vom Prinzip her um nichts weiter als um funktional erweiterte *anonyme Methoden*, wie es sie bereits seit C# 2.0 gibt. Der offensichtlichste Unterschied zeigt sich in der Syntax: auf die Parameterliste folgt ein Pfeil (=>), gefolgt von einer einzelnen Anweisung oder einem Anweisungsblock.
- Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Bereits C# 3.0 führte eine neue Syntax ein, die das direkte Hinzufügen neuer Methoden zu einer vorhandenen Klasse erlaubt. Diese so genannten *Erweiterungsmethoden* werden als statische Methoden in einer neuen statische Klasse implementiert und können dann wie eine normale Methode (d h. Instanzmethode) des erweiterten Datentyps aufgerufen werden.
- Nicht nur Zahlen-Auflistungen wie im vorliegenden Rezept, sondern jede Collection, die das *System.Collections.Generic.IEnumerable* Interface (oder das generische *IEnumerable<T>*-Interface) unterstützt, kann mit LINQ verarbeitet werden.

6.4.2 Aggregat-Abfragen mit LINQ

Im vorliegenden Praxisbeispiel demonstrieren wir, wie anhand von Aggregat-Abfragen eine Auflistung von Gleitkommazahlen (z.B. eine Messwertliste) nach verschiedenen Kriterien ausgewertet werden kann. Gewissermaßen als "Nebeneffekt" wird die Liste auch noch sortiert.

Oberfläche

Diesmal wollen wir das Array nicht fest kodieren, sondern dem Benutzer die Möglichkeit überlassen, eine beliebige Menge von Gleitkommazahlen über eine *TextBox* einzugeben. Die Zahlenkolonne wird in einer *ListBox* dargestellt und anschließend ausgewertet und sortiert angezeigt.



Quellcode

```
using System.Linq;
...
public partial class Form1 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
```

Die Länge der Zahlenliste wird bestimmt:

```
        int n = listBox1.Items.Count;
```

Die Zahlen werden aus der *ListBox* in ein *float*-Array übergeben:

```
        float[] zahlen = new float[n];
        for (int i = 0; i < zahlen.Length; i++)
            zahlen[i] = Convert.ToSingle(listBox1.Items[i]);
```

Die Abfrage wird definiert:

```
        var expr =
              from z in zahlen
              orderby z                              // sortiert in aufsteigender Reihenfolge
              select z;
```

Nacheinander wird der Abfrage-Ausdruck für verschiedene Aggregat-Methoden ausgewertet:

```
        int count = expr.Count();                    // Anzahl
        label1.Text = count.ToString();

        float sum = expr.Sum();                      // Summe
        label2.Text = sum.ToString();

        float avg = expr.Average();                  // Durchschnitt
        label3.Text = avg.ToString();

        float max = expr.Max();                     // Maximum
        label4.Text = max.ToString();

        float min = expr.Min();                     // Minimum
        label5.Text = min.ToString();
```

Sortierte Anzeige in *ListBox*:

```
        listBox1.Items.Clear();
        foreach (float z in expr) listBox1.Items.Add(z.ToString());
    }
```

ListBox-Inhalt löschen:

```
    private void button4_Click(object sender, EventArgs e)
    {
```

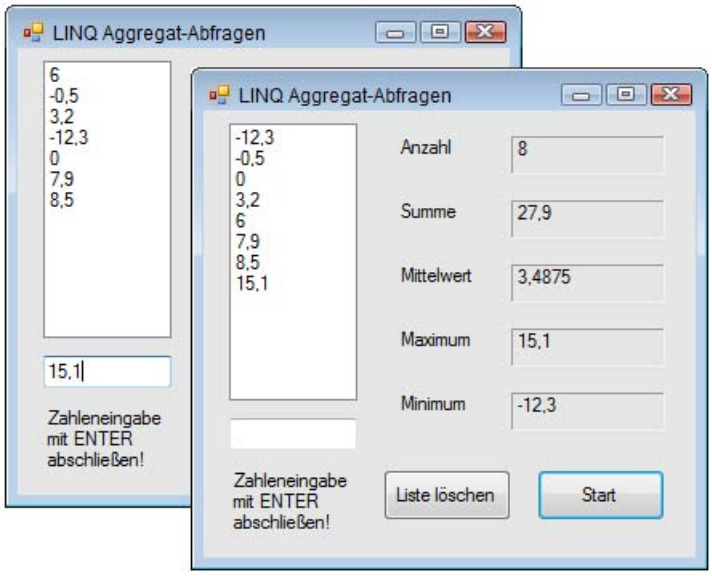
```
        listBox1.Items.Clear();
    }
```

Eine in die *TextBox* eingegebene Zahl wird mittels Enter-Taste in die *ListBox* übernommen:

```
    private void textBox1_KeyUp(object sender, KeyEventArgs e)
    {
        if (e.KeyCode == Keys.Enter)
        {
            listBox1.Items.Add(textBox1.Text);
            textBox1.Clear();
        }
    }
}
```

Test

Nach Eingabe einer Zahlenkolonne (Dezimaltrennzeichen ist natürlich das Komma!) können Sie Auswertung und Sortierung starten:



Bemerkungen

- Um die Zahlenreihe statt in aufsteigender, in absteigender Reihenfolge zu sortieren, ist im Abfrageausdruck die Klausel ***orderby z*** zu ersetzen durch ***orderby z descending***.
- Die Wirkung der Aggregatmethoden (*Count, Sum, Average, Min, Max*) ist völlig unabhängig davon, ob die Zahlenmenge sortiert ist oder nicht.

Teil

Teil II: Technologien

- **Zugriff auf das Dateisystem**
- **Dateien lesen und schreiben**
- **Asynchrone Programmierung**
- **Die Task Parallel Library**
- **Fehlersuche und Behandlung**

Kapitel 7

Zugriff auf das Dateisystem

Gewissermaßen als "Vorläufer" der Datenbanken dienen Dateien dazu, Daten auf Festplatte (oder anderen Speichermedien) dauerhaft zu sichern bzw. Daten zwischen einer Quelle und einem Ziel zu transportieren.

Inhalt des vorliegenden Kapitels ist aber zunächst nur die Arbeit auf Verzeichnisebene, womit das Erstellen, Löschen, Kopieren, Verschieben, Umbenennen, Durchsuchen und Überwachen von Verzeichnissen und Dateien gemeint ist.

HINWEIS: Zum Lesen und Abspeichern von Datei-Inhalten sowie zum Arbeiten mit Streams kommen wir erst im nachfolgenden Kapitel!

7.1 Grundlagen

Wenn Sie an Dateien denken, so fallen Ihnen dazu meist Begriffe wie Dateinamen, Dateigröße, Dateiattribute und Dateiverzeichnis ein. Als Programmierer sollten Sie aber auch den tieferen Sinn des Datei-Systems von Windows verstanden haben:

- Eine **Datei** unter Windows ist eine geordnete und benannte Sammlung von Byte-Folgen, die persistent (meist auf Festplatte) abgespeichert sind.
- Ein **Verzeichnis** (bzw. Ordner) in Windows ist einfach ein anderer Dateityp, der wiederum andere Dateien und Unterverzeichnisse enthalten kann.

In früheren Programmiersprachen waren Sie meist auf das *FileSystemObject* oder den Direktzugriff auf die Win32 API angewiesen, um Ihre Datei-Operationen zu implementieren.

HINWEIS: Das .NET-Framework stellt im Namespace *System.IO* eine Anzahl leistungsfähiger Klassen zur Verfügung, die den Zugriff auf das Windows-Dateisystem vereinfachen.

7.1.1 Klassen für den Zugriff auf das Dateisystem

Die für das vorliegende Kapitel interessanten Klassen des *System.IO*-Namespace entnehmen Sie der folgenden Tabelle.

Die wichtigsten Klassen für Manipulationen mit Verzeichnissen und Dateien sind die Pärchen *Directory/DirectoryInfo* und *File/FileInfo*, die sich vor allem hinsichtlich ihrer Instanziierbarkeit unterscheiden.

Klasse	Beschreibung
Directory	Die statischen Methoden erlauben das Erstellen, Verschieben und Benennen von Verzeichnissen und Unterverzeichnissen.
DirectoryInfo	Ähnelt der *Directory*-Klasse, enthält aber nur Instanz-Methoden.
Path	Die statischen Methoden erlauben die plattformübergreifende Arbeit mit Verzeichnissen.
File	Die statischen Methoden erlauben das Erzeugen, Kopieren, Löschen, Verschieben und Öffnen von Dateien.
FileInfo	Ähnelt der *File*-Klasse, enthält aber nur Instanz-Methoden.
FileSystemInfo	Basisklasse für *DirectoryInfo*- und *FileInfo*-Objekte
FileSystemWatcher	Löst Ereignisse zum Überwachen des Dateisystems aus.
DriveInfo	Liefert Laufwerksinformationen

Die Methoden der Klassen *File* und *FileInfo* liefern auch die Voraussetzungen für den Schreib- und Lesezugriff auf Dateien.

HINWEIS: Beachten Sie auch die Unter-Namensräume wie *System.IO.Compression* (Komprimieren von Daten) und *System.IO.Ports* (Zugriff auf die serielle Schnittstelle).

7.1.2 Statische versus Instanzen-Klasse

Bei **statischen Klassen** müssen Sie jeder Methode den Dateinamen oder den Verzeichnispfad übergeben. Das kann dann ziemlich lästig werden, wenn Sie diese Methoden öfters hintereinander aufrufen müssen. Die entsprechenden Eigenschaften der **Instanzen-Klassen** *FileInfo* und *DirectoryInfo* hingegen erlauben es Ihnen, den Datei- oder Verzeichnisnamen bereits im Konstruktor einmalig zu spezifizieren.

Ein weiterer wichtiger Unterschied zwischen beiden Klassen hat etwas mit der Performance zu tun:

HINWEIS: Wenn Sie mit den Methoden der statischen Klassen *File*, *Directory* und *Path* arbeiten, werden Sicherheitsprüfungen bei **jedem** Methodenaufruf vorgenommen, bei den Instanzen-Methoden der Klassen *FileInfo* und *DirectoryInfo* geschieht dies nur ein einziges Mal.

Wenn Sie also ein Objekt mehrmals wiederverwenden wollen, so nehmen Sie statt einer statischen Methode besser die entsprechende Instanzenmethode, da eine Sicherheitsprüfung nicht immer erforderlich ist.

BEISPIEL 7.1: Zwei alternative Möglichkeiten zum Anzeigen von Erstellungsdatum und Zeitpunkt des letzten Zugriffs auf die Datei *c:\test\info.txt*

C#

```
using System.IO;
```

Mit statischer Klasse:

```
label1.Text = File.GetCreationTime(@"c:\test\info.txt").ToString();
label2.Text = File.GetLastAccessTime(@"c:\test\info.txt").ToString();
```

oder mit Instanzen-Klasse:

```
FileInfo myFile = new FileInfo(@"c:\test\info.txt");
label1.Text = myFile.CreationTime.ToString();
label2.Text = myFile.LastAccessTime.ToString();
```

7.2 Übersichten

Die folgenden Zusammenstellungen sind keinesfalls vollständig und sollen Ihnen lediglich die Orientierung im Gewirr der vielfältigen Methoden und Eigenschaften erleichtern.

HINWEIS: Zu praktischen Beispielen kommen wir in den Abschnitten 7.3 und 7.7.

7.2.1 Methoden der Directory-Klasse

Die folgende Tabelle gibt eine Übersicht über wichtige statische Methoden:

Methode	Beschreibung
CreateDirectory	... erzeugt ein Verzeichnis oder ein Unterverzeichnis
Delete	... löscht ein Verzeichnis
Exists	... überprüft die Existenz eines Verzeichnisses
GetCreationTime	... liefert Zeitpunkt der Erstellung
GetDirtectories	... liefert die Namen aller Unterverzeichnisse eines bestimmten Ordners
GetFiles	... liefert alle Dateinamen eines bestimmten Ordners
GetParent	... liefert den Namen des übergeordneten Verzeichnisses
Move	... verschiebt ein Verzeichnis inkl. Dateien in ein anderes Verzeichnis
SetCreationTime	... legt den Zeitpunkt der Erstellung eines Verzeichnisses fest

7.2.2 Methoden eines DirectoryInfo-Objekts

In der folgenden Tabelle wurden einige wichtige Methoden weggelassen, da es diese mit gleicher bzw. ähnlicher Bedeutung bereits in der *Directory*-Klasse gibt.

Methode	Beschreibung
CreateSubdirectory	... erstellt ein oder mehrere Unterverzeichnis(se) im angegebenen Pfad (Pfad kann relativ sein)
GetFileSystemInfos	... ruft ein Array von *FileSystemInfo*-Objekten mit Dateien und Unterverzeichnissen ab
MoveTo	... verschiebt ein *DirectoryInfo*-Objekt nebst Inhalt in einen neuen Pfad

7.2.3 Eigenschaften eines DirectoryInfo-Objekts

Die meisten der folgenden Eigenschaften wurden von der Basisklasse *FileSystemInfo* geerbt und finden sich deshalb auch beim *FileInfo*-Objekt.

Eigenschaft	Beschreibung
CreationTime	... liest oder schreibt den Erstellungszeitpunkt des Verzeichnisses
Exists	... liest einen Wert, der angibt, ob das Verzeichnis vorhanden ist
FullName	... liest den vollständigen Pfad des Verzeichnisses
LastAccessTime	... liest oder schreibt den Zeitpunkt des letzten Zugriffs auf das aktuelle Verzeichnis
LastWriteTime	... liest oder schreibt den Zeitpunkt des letzten Schreibzugriffs auf das aktuelle Verzeichnis
Parent	... liest das übergeordnete Verzeichnis eines angegebenen Unterverzeichnisses
Root	... liest das Stammverzeichnis eines Pfads

7.2.4 Methoden der File-Klasse

Die folgende Tabelle gibt eine Übersicht über wichtige statische Methoden:

Methode	Typ	Beschreibung
AppendAllText	*void*	... öffnet die Datei und hängt eine Zeichenfolge an
AppendText	*StreamWriter*	... hängt Text an eine vorhandene Datei an
Copy	*void*	... kopiert eine Datei an einen anderen Speicherort
Create	*FileStream*	... erzeugt eine Datei in einem bestimmten Pfad
CreateText	*StreamWriter*	... erzeugt und öffnet eine Textdatei
Delete	*void*	... löscht eine Datei

Methode	Typ	Beschreibung
Exists	*Boolean*	... prüft die Existenz einer Datei
GetAttributes	*FileAttributes*	... liefert die Dateiattribute
GetCreationTime *GetLastAccessTime* *GetLastWriteTime*	*DateTime*	... liefert Zeitpunkt der Erstellung, ... des letzten Zugriffs, ... des letzten Schreibzugriffs
Move	*void*	... verschiebt eine Datei oder benennt sie um
Open	*FileStream*	... öffnet eine Datei
OpenRead	*FileStream*	... öffnet eine Datei zum Lesen.
OpenText	*StreamReader*	... öffnet eine Textdatei zum Lesen
OpenWrite	*FileStream*	... öffnet eine Datei zum Schreiben
ReadAllBytes	*byte[]*	... öffnet Binärdatei und liest Inhalt in in Byte-Array
ReadAllLines	*string[]*	... öffnet Textdatei und liest alle Zeilen in String-Array
ReadAllText	*string*	... öffnet Textdatei, liest alle Zeilen in einen String ein und schließt Datei
SetAttributes	*void*	... setzt die Dateiattribute
SetCreationTime *SetLastAccesTime* *SetLastWriteTime*	*void*	... setzt Zeitpunkt der Erstellung, ... des letzten Zugriffs, ... des letzten Schreibzugriffs
WriteAllBytes	*void*	... erzeugt neue Datei und schreibt Byte-Array hinein
WriteAllLines	*void*	... erzeugt neue Datei und schreibt String-Array hinein
WriteAllText	*void*	... erzeugt neue Datei und schreibt String hinein

7.2.5 Methoden eines FileInfo-Objekts

Viele Methoden des *FileInfo*-Objekts haben gleichnamige statische Pendants in der *File*-Klasse.

Methode	Typ	Beschreibung
AppendText	*StreamWriter*	... fügt Text einer vorhandenen Datei hinzu
CopyTo	*FileInfo*	... kopiert die Datei in ein anderes Verzeichnis
Create	*FileStream*	... erzeugt eine Datei
CreateText	*StreamWriter*	... erzeugt eine Textdatei
Delete	*void*	... löscht eine Datei
Exists	*Boolean*	... prüft das Vorhandensein einer Datei
MoveTo	*void*	... verschiebt die Datei oder benennt sie um
Open	*FileStream*	... öffnet eine Datei

Methode	Typ	Beschreibung
OpenRead	*FileStream*	... öffnet eine Datei für den Lesezugriff
OpenText	*StreamReader*	... öffnet eine Textdatei für den Lesezugriff
OpenWrite	*FileStream*	... öffnet eine Datei für den Schreibzugriff

7.2.6 Eigenschaften eines FileInfo-Objekts

Die meisten der folgenden Eigenschaften von *FileInfo* finden Sie ebenfalls beim *DirectoryInfo*-Objekt, da beide von der Basisklasse *FileSystemInfo* abgeleitet sind:

Eigenschaft	Beschreibung
Attributes	... liest oder schreibt die Dateiattribute (*ReadOnly*, *Hidden* usw.)
CreationTime	... liest oder schreibt Zeitpunkt der Erstellung einer Datei
Directory	... liefert Instanz des übergeordneten Verzeichnisses
DirectoryName	... liefert den vollständigen Dateipfad
Extension	... liefert Dateiextension (z.B. *txt* für Textdateien)
FullName	... liefert den vollständigen Dateipfad plus Dateinamen
LastAccessTime	... liest oder schreibt den Zeitpunkt des letzten Zugriffs
LastWriteTime	... liest oder schreibt den Zeitpunkt des letzten Schreibzugriffs
Length	... liefert die Dateigröße in Bytes
Name	... liefert den vollständigen Dateinamen

7.3 Operationen auf Verzeichnisebene

Wir behandeln hier die Existenzprüfung, das Erstellen, Löschen, Kopieren, Verschieben, Umbenennen, Durchsuchen und Überwachen von Dateien und Verzeichnissen.

HINWEIS: Bei allen folgenden Beispielen ist der Namespace *System.IO* einzubinden!

7.3.1 Existenz eines Verzeichnisses/einer Datei feststellen

Für diese häufig zu lösende Aufgabenstellung eignet sich die statische *Exists*-Methode der *Directory*- bzw. *File*-Klasse (auch *DirectoryInfo*- und *FileInfo*-Objekte verfügen über diese Methode):

BEISPIEL 7.2: Die Existenz einer Datei feststellen

C#
```
string pfad = @"c:\TestXY.txt";
if (!File.Exists(pfad))
       throw new FileNotFoundException();
```

BEISPIEL 7.2: Die Existenz einer Datei feststellen

Ergebnis

Der Code zur Existenzprüfung eines Verzeichnisses ist äquivalent, nur dass Sie diesmal die *Directory*-Klasse verwenden und im Fehlerfall eine *DirectoryNotFound*-Exception ausgelöst wird.

7.3.2 Verzeichnisse erzeugen und löschen

Die einfachsten Möglichkeiten zum Erzeugen und Löschen von Verzeichnissen bieten die Methoden *CreateDirectory* und *Delete* der *Directory*-Klasse.

BEISPIEL 7.3: Ein Verzeichnis erzeugen und anschließend wieder löschen

C#

```
string pfad = @"c:\temp";
Directory.CreateDirectory(pfad);   // falls Verzeichnis bereits vorhanden, passiert nichts!
Directory.Delete(pfad, true);      // löscht auch vorhandene Unterverzeichnisse und Dateien
```

Das gleiche Ziel, allerdings etwas umständlicher, erreicht man mit der *Create*-Methode der instanziierbaren *DirectoryInfo*-Klasse, wobei über *CreateSubdirectory* auch das Hinzufügen von Unterverzeichnissen möglich ist.

BEISPIEL 7.4: Ein Verzeichnis und ein Unterverzeichnis anlegen und wieder löschen

C#

```
DirectoryInfo di = new DirectoryInfo(@"c:\temp");
di.Create();
di.CreateSubdirectory("temp1");
di.Delete(true);            // löscht inklusive vorhandener Unterverzeichnisse und Dateien
```

HINWEIS: Der Aufruf von *Delete* ohne Parameterangabe ist nur erfolgreich, wenn das Verzeichnis leer ist!

7.3.3 Verzeichnisse verschieben und umbenennen

Für diese Aufgaben verwenden Sie am besten die *Move*-Methode der *Directory*-Klasse.

Leider gibt es sowohl in der *Directory*- als auch in der *DirectoryInfo*-Klasse noch keine Methode, die das Kopieren eines kompletten Verzeichnisses ermöglicht.

BEISPIEL 7.5: Verzeichnis *c:\tempX* nach *c:\beispiele* verschieben und umbenennen

C#
```
Directory.Move(@"c:\tempX", @"c:\beispiele\tempY");
```

7.3.4 Aktuelles Verzeichnis bestimmen

Verwenden Sie dazu die *GetCurrentDirectory*- bzw. *SetCurrentDirectory*-Methode der *Directory*-Klasse.

BEISPIEL 7.6: Festlegen und Anzeigen des aktuellen Arbeitsverzeichnisses

C#
```
Directory.SetCurrentDirectory(@"c:\test");
label1.Text = Directory.GetCurrentDirectory();     // zeigt "c:\test"
```

Wenn der Dateiname **ohne** Pfad angegeben wird, bezieht sich die Datei automatisch auf das Projekt- bzw. Arbeitsverzeichnis.

BEISPIEL 7.7: Im Projektverzeichnis wird ein Verzeichnis *\temp* angelegt

C#
```
Directory.CreateDirectory("temp");
```

7.3.5 Unterverzeichnisse ermitteln

Um **alle** Unterverzeichnisse zu ermitteln, verwenden Sie die *GetDirectories*-Methode der *DirectoryInfo*-Klasse.

BEISPIEL 7.8: Es werden alle Unterverzeichnisse von *c:* in einer *ListBox* angezeigt

C#
```
DirectoryInfo myDir = new DirectoryInfo(@"c:\");       // neues DirectoryInfo-Objekt
DirectoryInfo[] myDirs = myDir.GetDirectories();       // Unterverzeichnisse ermitteln
                                                       // und im Array ablegen
for (int i = 0; i < myDirs.Length; i++)   // alle Unterverzeichnisse durchlaufen ...
     listBox1.Items.Add(myDirs[i].Name); // ... und Verzeichnisnamen zur ListBox hinzufügen
```

Eine alternative Lösung bietet sich mit der gleichnamigen Methode der *Directory*-Klasse:

BEISPIEL 7.9: Lösung des gleichen Problems wie im Vorgängerbeispiel

C#
```
string[] myDirs = Directory.GetDirectories(@"c:\");
for (int i = 0; i < myDirs.Length; i++)
          listBox1.Items.Add(myDirs[i]);
```

7.3.6 Alle Laufwerke ermitteln

Die Klasse *DriveInfo* ermöglicht den Zugriff auf diverse Laufwerksinformationen. Alles Weitere soll direkt am Beispiel erklärt werden.

BEISPIEL 7.10: Auflisten aller Laufwerke des Systems inklusive dazugehöriger Informationen

C#

```
foreach (DriveInfo di in DriveInfo.GetDrives())
{
    listBox1.Items.Add("Laufwerk: " + di.Name);
    if (di.IsReady)
    {
        listBox1.Items.Add("Bezeichnung: " + di.VolumeLabel);
        listBox1.Items.Add("Typ: " + di.DriveType);
        listBox1.Items.Add("Bezeichnung: " + di.DriveFormat);
        listBox1.Items.Add("Größe: " + di.TotalSize);
        listBox1.Items.Add("Freier Platz: " + di.TotalFreeSpace);
        listBox1.Items.Add("---------------------------------");
    }
    else
    {
        listBox1.Items.Add("Laufwerk ist nicht bereit!");
        listBox1.Items.Add("---------------------------------");
     }
}
```

Ergebnis

7.3.7 Dateien kopieren und verschieben

Am einfachsten realisieren Sie diese Aufgabe mit den statischen *Copy*- bzw. *Move*-Methoden der *File*-Klasse.

BEISPIEL 7.11: Datei kopieren und anschließend verschieben

C#
```
string sourcePath = @"c:\sample.txt" ;
string destPath = @"c:\sample1.txt";
string movePath = @"c:\temp\sample1.txt";
File.Copy(sourcePath, destPath);          // kopieren
File.Move(sourcePath, movePath);          // verschieben
```

Falls Sie lieber mit Instanzen arbeiten, können Sie die Methoden *CopyTo* und *MoveTo* der Klasse *FileInfo* verwenden.

BEISPIEL 7.12: Obiges Beispiel wird mit Methoden der *FileInfo*-Klasse realisiert

C#
```
string sourcePath = @"c:\sample.txt" ;
string destPath = @"c:\sample1.txt";
string movePath = @"c:\temp\sample1.txt";

FileInfo myFile = new FileInfo(sourcePath);
myFile.CopyTo(destPath);          // kopieren
myFile.MoveTo(movePath);          // verschieben
```

7.3.8 Dateien umbenennen

Leider bietet das .NET-Framework keinerlei Möglichkeit zum direkten Umbenennen einer Datei, da die *Name*-Eigenschaft der *FileInfo*-Klasse schreibgeschützt ist und eine *Rename*-Methode fehlt. Verwenden Sie deshalb zum Umbenennen die Methoden *Move* der Klasse *File* bzw. *MoveTo* der Klasse *FileInfo*.

BEISPIEL 7.13: Umbenennen der Datei *info.txt* in *info_1.txt*

C#
```
FileInfo myFile = new FileInfo(@"c:\test\info.txt");
myFile.MoveTo(@"c:\test\info_1.txt");
```

7.3.9 Dateiattribute feststellen

Um die Dateiattribute zu ermitteln, kann man entweder auf die Eigenschaften der *FileInfo*-Klasse, oder aber auch auf die entsprechenden (statischen) Methoden der *File*-Klasse zugreifen:

Eigenschaft *FileInfo*-Klasse	Methode *File*-Klasse	Beschreibung
Attributes	*GetAttributes()* *SetAttributes()*	Wert basiert auf Dateiattribute-Flags (*Archive*, *Compressed*, *Directory*, *Hidden* ...)
CreationTime	*GetCreationTime()* *SetCreationTime()*	Datum/Uhrzeit der Erstellung
LastAccessTime	*GetLastAccessTime()* *SetLastAccessTime()*	Datum/Uhrzeit des letzten Zugriffs

Eigenschaft *FileInfo*-Klasse	Methode *File*-Klasse	Beschreibung
LastWriteTime	*GetLastWriteTime()* *SetLastWriteTime()*	Datum/Uhrzeit des letzten Schreibzugriffs
Exists	*Exists()*	Liefert *true*, falls Datei physikalisch existiert

BEISPIEL 7.14: Anzeige des Erstellungsdatums einer Datei

C#
```
label1.Text = File.GetCreationTime("Liesmich.txt").ToString();
```

BEISPIEL 7.15: Feststellen, ob Datei im Arbeitsverzeichnis existiert

C#
```
if (File.Exists("Liesmich.txt")) MessageBox.Show("Datei ist vorhanden!");
```

oder

```
FileInfo myFile = new FileInfo("Liesmich.txt");
if (myFile.Exists) MessageBox.Show("Datei ist vorhanden!");
```

Attribute von Verzeichnissen und Dateien

Die verschiedenen Attribute für Dateien und Verzeichnisse sind in der *FileAttribute*-Enumeration anzutreffen. Die folgende Tabelle zeigt die wichtigsten:

Mitglied	Beschreibung
Archive	Archiv-Status der Datei, wie er häufig zum Markieren einer zu löschenden oder einer Backup-Datei verwendet wird
Compressed	Entspricht einer gepackten Datei
Directory	Die Datei ist in Wirklichkeit ein Verzeichnis
Encrypted	Die Datei ist verschlüsselt
Hidden	Die Datei ist versteckt und in einem gewöhnlichen Verzeichnis unsichtbar
Normal	Es wurden keine Datei-Attribute gesetzt
ReadOnly	Die Datei kann nicht verändert, sondern nur gelesen werden
System	Die Datei gehört zum Betriebssystem oder wird exklusiv von diesem benutzt
Temporary	Die Datei ist temporär, sie wird vom Programm angelegt und wieder gelöscht

Um das Vorhandensein eines bestimmten Datei-Attributes festzustellen, muss eine bitweise ODER-Verknüpfung durchgeführt werden.

BEISPIEL 7.16: In einer *CheckBox* wird angezeigt, ob es sich um eine Archiv-Datei handelt

C#
```
FileAttributes attbs = File.GetAttributes(@"c:\beispiele\test.dat");
if(attbs==(attbs|FileAttributes.Archive))
   checkBox2.Checked = true;
```

BEISPIEL 7.16: In einer *CheckBox* wird angezeigt, ob es sich um eine Archiv-Datei handelt

```
else
   checkBox2.Checked = false;
```

7.3.10 Verzeichnis einer Datei ermitteln

BEISPIEL 7.17: Das Verzeichnis der Datei *liesmich.txt* wird in einem Label angezeigt

C#
```
FileInfo myFile = new FileInfo("Liesmich.txt");
label1.Text = myFile.DirectoryName;
```

7.3.11 Alle im Verzeichnis enthaltenen Dateien ermitteln

Gewissermaßen als Ergänzung zu *GetDirectories* können Sie mit der *GetFiles*-Methode der Klasse *DirectoryInfo* alle in einem Verzeichnis enthaltenen Dateien ermitteln.

BEISPIEL 7.18: Alle im Verzeichnis *c:|* abgelegten Dateien werden in einer *ListBox* angezeigt.

C#
```
DirectoryInfo myDir = new DirectoryInfo(@"c:\");       // neues DirectoryInfo-Objekt
FileInfo[] myFiles = myDir.GetFiles();         // Dateien ermitteln und im Array ablegen
for (int i = 0; i < myFiles.Length; i++)       // alle Dateien durchlaufen ...
     listBox1.Items.Add(myFiles[i].Name);      // ... und Dateinamen zur ListBox hinzufügen
```

Noch kürzer ist der Code bei Verwendung der (statischen) *Directory*-Klasse.

BEISPIEL 7.19: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst

C#
```
string[] myFiles = Directory.GetFiles(@"c:\");   // String-Array füllen ...
for (int i = 0; i < myFiles.Length; i++)         // alle Einträge durchlaufen
     listBox1.Items.Add(myFiles[i]);             // ... und anzeigen
```

Sollen nicht nur die Dateien im angegebenen Verzeichnis, sondern auch die in allen Unterverzeichnissen vorhandenen Dateien mit angezeigt werden, so sind die Suchoptionen zu spezifizieren.

BEISPIEL 7.20: Es werden alle *.htm*-Dateien im Verzeichnis *c:|doberenz* und alle darunter liegenden Verzeichnisse angezeigt

C#
```
string topDir = @"c:\doberenz";
string[] myFiles = Directory.GetFiles(topDir, "*.htm", SearchOption.AllDirectories);
for (int i = 0; i < myFiles.Length; i++)         // alle Einträge durchlaufen
     listBox1.Items.Add(myFiles[i]);             // ... und anzeigen
```

7.3.12 Dateien und Unterverzeichnisse ermitteln

Im Zusammenhang mit der *Directory*-Eigenschaft der *FileInfo*-Klasse verdient die *GetFileSystemInfos*-Methode der *DirectoryInfo*-Klasse unsere besondere Aufmerksamkeit.

BEISPIEL 7.21: In einer *TextBox* (*MultiLine* = *true*) werden das Verzeichnis einer Datei und alle weiteren sich im gleichen Verzeichnis befindlichen Dateien und Unterverzeichnisse angezeigt.

C#

Datei im Anwendungsverzeichnis öffnen oder neu erzeugen:

```
FileInfo myFile = new FileInfo("Test.txt");
```

Verweis auf Verzeichnis erzeugen:

```
DirectoryInfo myDir = myFile.Directory;
```

Alle Einträge ermitteln:

```
FileSystemInfo[] fsi = myDir.GetFileSystemInfos();
```

Vollständigen Pfad anzeigen:

```
textBox1.Text = myDir.FullName + Environment.NewLine;
```

... und alle weiteren Unterverzeichnisse und Dateien:

```
foreach (FileSystemInfo info in fsi)
        textBox1.Text += info.Name + Environment.NewLine;
```

HINWEIS: Wer seine Erkundungen zu Datei- und Verzeichnisoperationen mit einem komplexeren Beispiel abrunden möchte, dem sei das Praxisbeispiel 7.7.1 "Infos über Verzeichnisse und Dateien gewinnen" nahegelegt, an welchem er viele der in diesem Abschnitt besprochenen Features der *FileInfo*-Klasse noch einmal im Zusammenspiel erleben kann.

7.4 Zugriffsberechtigungen

Die Klassenpärchen *Directory-/DirectoryInfo*- und *File-/FileInfo* verfügen über die *GetAccessControl*-Methode. Diese liefert ein *DirectorySecurity*- *bzw. FileSecurity*-Objekt, welches die *Access Control List* (ACL) der Datei bzw. des Verzeichnisses kapselt (Namespace *System.Security.AccessControl*).

7.4.1 ACL und ACE

Eine ACL enthält *Access Control Entries* (ACE), welche die Zugriffsregeln von Benutzern oder Benutzergruppen zur Ausführung spezieller Aktionen für eine gegebene Datei oder ein Verzeichnis beschreiben.

Die *AddAccessRule*-Method des *DirectorySecurity*- bzw. *FileSecurity*-Objekts fügt eine neue ACE zur vorhandenen ACL hinzu. Analog dazu entfernt die *RemoveAccessRule*-Methode diese Zugriffsregel wieder.

Eine Zugriffsregel wird in einer Instanz der *FileSystemAccessRule*-Klasse gekapselt, sie spezifiziert den User Account, den Zugriffstyp (read, write usw.) und ob dieses Recht gewährt oder abgelehnt wird.

HINWEIS: Im Namespace *System.Security.AccessControl* finden sich zahlreiche weitere Klassen zum Verwalten von Zugriffsberechtigungen (ACLs).

7.4.2 SetAccessControl-Methode

Um eine neue oder geänderte ACL-Information dauerhaft für ein Verzeichnis oder eine Datei zu speichern verwenden Sie die *SetAccessControl*-Methode.

***BEISPIEL 7.22: SetAccessControl*-Methode**

C#

Ein neues Verzeichnis wird angelegt. Jedem Benutzer werden die uneingeschränkten Zugriffsrechte auf dieses Verzeichnisses verwehrt. Es wird also nicht gelingen, das angelegte Verzeichnis zu löschen.

```
using System.IO;
using System.Security.AccessControl;
...
private string dir = "Testverzeichnis";

Directory.CreateDirectory(dir);
string account = "Jeder";
DirectorySecurity dsc = Directory.GetAccessControl(dir);
FileSystemRights rights = FileSystemRights.FullControl;
AccessControlType controlType = AccessControlType.Deny;
FileSystemAccessRule fsar = new FileSystemAccessRule(account, rights, controlType);

dsc.AddAccessRule(fsar);                    // Zugriffsrechte hinzufügen
Directory.SetAccessControl(dir, dsc);       // ... und speichern
```

7.4.3 Zugriffsrechte anzeigen

Die einzelnen Zugriffsrechte (ACEs) sind in der *AuthorizationRuleCollection* des *DirectorySecurity- bzw. FileSecurity*-Objekts enthalten.

BEISPIEL 7.23: Die ACL des im Vorgängerbeispiel erzeugten Verzeichnisses wird ausgelesen, die einzelnen ACEs werden in einer *ListBox* angezeigt.

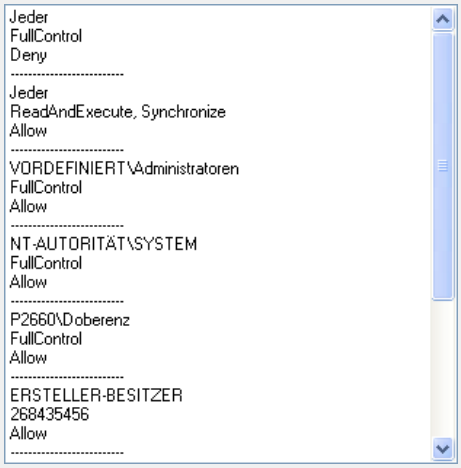
C#

```
using System.IO;
using System.Security.AccessControl;
...
private string dir = "Testverzeichnis";
DirectorySecurity fsc = Directory.GetAccessControl(dir);

AuthorizationRuleCollection arcoll = fsc.GetAccessRules(true, true,
                                        typeof(System.Security.Principal.NTAccount));
foreach (FileSystemAccessRule fsar in arcoll)
{
    listBox1.Items.Add(fsar.IdentityReference.ToString());
```

BEISPIEL 7.23: Die ACL des im Vorgängerbeispiel erzeugten Verzeichnisses wird ausgelesen, die einzelnen ACEs werden in einer *ListBox* angezeigt.

```
        listBox1.Items.Add(fsar.FileSystemRights.ToString());
        listBox1.Items.Add(fsar.AccessControlType.ToString());
        listBox1.Items.Add("--------------------------");
    }
```

Ergebnis



7.5 Weitere wichtige Klassen

Mit *Path* und *FileSystemWatcher* sollen die Ausführungen dieses Kapitels abgerundet werden.

7.5.1 Die Path-Klasse

Als Alternative zur *Directory*-Klasse kommt auch die *Path*-Klasse infrage. Allerdings ist hier Vorsicht geboten, denn die meisten Mitglieder der *Path*-Klasse wirken nicht mit dem Dateisystem zusammen und überprüfen deshalb nicht, ob die durch eine Pfadzeichenfolge angegebene Datei auch tatsächlich vorhanden ist.

Methode	Beschreibung
GetDirectoryName	... extrahiert aus dem übergebenen Dateipfad das Verzeichnis
GetExtension	... liefert aus dem übergebenen Dateipfad die Extension[1]
GetFileName	... liefert den vollständigen Dateinamen
GetFileNameWithoutExtension	... liefert Dateinamen ohne Extension
GetFullPath	... liefert den kompletten Pfadnamen
GetPathRoot	... liefert das Wurzelverzeichnis

[1] ... einschließlich des führenden Punkts

BEISPIEL 7.24: Ausgabe von Datei-Informationen in einer *ListBox*

C#
```
using System.IO;
...
string verz = @"c:\test\info.txt";
listBox1.Items .Add("Verzeichnis : " + System.IO.Path.GetDirectoryName(verz));
listBox1.Items.Add("Dateiname : " + System.IO.Path.GetFileName(verz));
listBox1.Items.Add("Dateiname ohne Extension : " +
                    System.IO.Path.GetFileNameWithoutExtension(verz));
listBox1.Items.Add("Dateiextension : " + System.IO.Path.GetExtension(verz));
listBox1.Items.Add("Rootverzeichnis : " + System.IO.Path.GetPathRoot(verz));
listBox1.Items.Add("Temporäres Verzeichnis : " + System.IO.Path.GetTempPath());
listBox1.Items.Add("Neues Tempfile : " + System.IO.Path.GetTempFileName());
```

7.5.2 Die Klasse FileSystemWatcher

Die Klasse *FileSystemWatcher* dient dem einfachen Beobachten des Dateisystems, sie löst beispielsweise Ereignisse aus, wenn Dateien oder Verzeichnisse geändert werden.

Die folgende Tabelle zeigt einige wichtige Eigenschaften:

Eigenschaft	Beschreibung
NotifyFilter	Typ der zu überwachenden Änderung
Filter	Filterzeichenfolge für die zu überwachenden Dateien
EnableRaisingEvents	Aktivieren der Komponente (*true*/*false*)

Die verschiedenen Werte der *NotifyFilter*-Eigenschaft sind in der *NotifyFilters*-Enumeration enthalten:

Mitglied	Typ der zu überwachenden Änderung
Attributes	Attribute der Datei oder des Verzeichnisses
CreationTime	Zeitpunkt der Erstellung der Datei oder des Verzeichnisses
DirectoryName	Name des Verzeichnisses
FileName	Name der Datei
LastAccess	Datum des letzten Öffnens der Datei oder des Verzeichnisses
LastWrite	Datum des letzten Schreibzugriffs auf die Datei oder das Verzeichnis
Security	Sicherheitseinstellungen der Datei oder des Verzeichnisses
Size	Größe der Datei oder des Verzeichnisses

Die folgende Tabelle zeigt einige wichtige Ereignisse

Ereignis	... tritt ein wenn im übergebenen Pfad ...
Changed	... eine Datei oder ein Verzeichnis geändert wird
Created	... eine Datei oder ein Verzeichnis erzeugt wird
Deleted	... eine Datei oder ein Verzeichnis gelöscht wird
Renamed	... eine Datei oder ein Verzeichnis umbenannt wird

BEISPIEL 7.25: Überwachen von vier Ereignissen von *.txt*-Dateien im Verzeichnis *c:|Beispiele*

C#
```
using System.IO;
...
FileSystemWatcher watcher = new FileSystemWatcher(@"C:\Beispiele");
watcher.NotifyFilter = (NotifyFilters.LastAccess | NotifyFilters.FileName);
watcher.Filter = "*.txt";
watcher.Changed += new  FileSystemEventHandler(OnChanged);    // Datei wurde geändert
watcher.Created +=  new FileSystemEventHandler(OnChanged);    //... neu hinzugefügt
watcher.Deleted += new FileSystemEventHandler(OnChanged);     //... gelöscht
watcher.Renamed += new RenamedEventHandler(OnRenamed);        //... umbenannt
watcher.SynchronizingObject = this;
watcher.EnableRaisingEvents = true;                       // Start der Überwachung
```

Die beiden folgenden Event-Handler spezifizieren die Reaktion auf die vier Ereignisse, die Anzeige erfolgt in einer *ListBox*:

```
private void OnChanged(object Source, FileSystemEventArgs e)
{
   listBox1.Items.Add("Datei: " + e.FullPath + " " + e.ChangeType.ToString());
}

private void OnRenamed(object Source, RenamedEventArgs e)
{
   listBox1.Items.Add("Datei: " + e.OldFullPath + " umbenannt in " + e.FullPath);
}
```

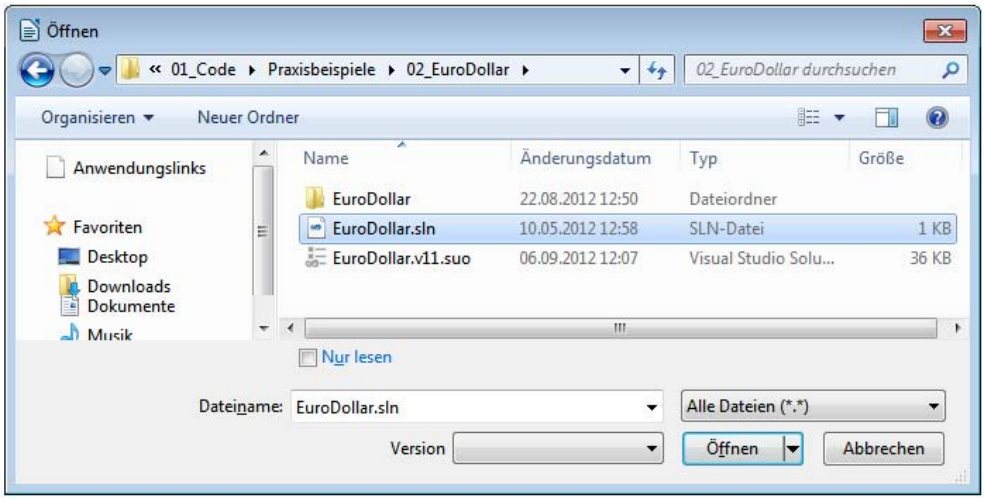
7.6 Datei- und Verzeichnisdialoge

Fast jedes Windows-Programm hat Menüoptionen zum Laden (Öffnen) bzw. Sichern (Speichern) von Dateien oder auch nur zur Verzeichnisauswahl. Sollten Sie einen solchen Dialog schon einmal "zu Fuß" programmiert haben, so wissen Sie, welcher Aufwand dafür nötig ist. .NET bietet Ihnen dafür die Controls *OpenFileDialog*, *SaveFileDialog* oder *FolderBrowserDialog*, welche direkt die Windows-Ressourcen "anzapfen". Ihre eigenen Programmschöpfungen erhalten damit die gleichen Dialoge wie auch andere "professionelle" Windows-Applikationen.

7.6.1 OpenFileDialog und SaveFileDialog

Anzeige und Auswertung

Durch Aufruf der Methode *ShowDialog* wird die entsprechende Dialogbox angezeigt. Über den Rückgabewert können Sie auswerten, welche Taste (OK/Abbruch) der Nutzer gedrückt hat, die Eigenschaft *FileName* enthält den Namen und den kompletten Pfad der ausgewählten Datei.



BEISPIEL 7.26: Auswahl einer einzelnen Datei

C#
```
string dateiname = String.Empty;
if (openFileDialog1.ShowDialog() ==  DialogResult.OK)
       dateiname = openFileDialog1.FileName;
```

BEISPIEL 7.27: Auswahl von mehreren Dateien (*MultiSelect=True*)

C#
```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
      for (int i = 0; i <= openFileDialog1.FileNames.GetUpperBound(0); i++)
                  MessageBox.Show(openFileDialog1.FileNames[i]);
```

Wichtige Eigenschaften

Eigenschaft	Beschreibung
Title	Die Beschriftung des Dialogfeldes
FileName	Das Ausgangsverzeichnis bzw. ein bereits vorgegebener Dateiname
Filters	Über diese Eigenschaft werden alle Dateifilter für das Dialogfeld bestimmt
FilterIndex	... legt den Standardfilter beim Öffnen des Dialogs fest
InitialDirectory	... legt das Startverzeichnis fest

Eigenschaft	Beschreibung
MultiSelect	Mit *True* legen Sie fest, dass mehr als eine Datei markiert werden kann
CheckFileExists	... prüft, ob die Datei auch physikalisch vorhanden ist
CheckPathExits	... prüft, ob der Pfad vorhanden ist
DereferenceLinks	... bestimmt, ob Hyperlinks aufgelöst werden, d.h., ob der Ursprungsdateiname zurückgegeben wird

Dateifilter

Die *Filter*-Eigenschaft legt die Dateitypen fest, die im Feld "Dateityp" zur Auswahl stehen. Die Syntax bietet einige Besonderheiten:

SYNTAX: `Dialog.Filter[ = descr1 |filter1 |descr2 |filter2 ...]`

Wählen Sie z.B. den Filter **.TXT*, so werden nur Textfiles angezeigt. Als Separator zwischen der Beschreibung (*descr*) und dem Filter wird der senkrechte Strich verwendet.

HINWEIS: Vor und nach diesem Trennstrich dürfen keine Leerzeichen stehen!

BEISPIEL 7.28: Mehrfachauswahl

C#
```
openFileDialog1.Filter = "Text (*.txt)|*.txt|Bild(*.bmp;*.ico)|*.bmp;*.ico";
```

Dieser Filter erlaubt es, sowohl Textdateien als auch Grafiken (Bitmaps und Icons) zur Anzeige auszuwählen. Die *FilterIndex*-Eigenschaft legt fest, welcher Filter der aktuelle ist.

7.6.2 FolderBrowserDialog

Geht es nur um die Auswahl eines Verzeichnisses, so ist dieser einfache Dialog angesagt. Die Funktionalität ähnelt (stark abgerüstet) obigen Dateidialogen.

BEISPIEL 7.29: Verzeichnisauswahl (Start ist der Ordner des Benutzers)

C#
```
folderBrowserDialog1.RootFolder = System.Environment.SpecialFolder.UserProfile;
DialogResult result = this.folderBrowserDialog1.ShowDialog();
if (result == DialogResult.OK)
{
    startFolder = this.folderBrowserDialog1.SelectedPath;
    label1.Text = startFolder;              // Anzeige des ausgewählten Verzeichnisses
}
```

Die SpecialFolder-Enumeration

Im obigen Beispiel wurde das Startverzeichnis (*RootFolder*) des *FolderBrowserDialog*s mit einer Konstanten der *SpecialFolder*-Enumeration festgelegt. Mit dieser Enumeration aus der Klasse

System.Environment haben Sie bequemen Zugriff auf Windows-spezifische Verzeichnisse bzw. Ordner.

HINWEIS: Eine nahezu äquivalente Alternative wäre der Einsatz der Klasse *SpecialDirectories*, dazu ist aber ein Verweis auf *Microsoft.VisualBasic.dll* erforderlich (Namespace *Microsoft.VisualBasic.FileIO*).

Die folgende Tabelle zeigt eine kleine Auswahl aus einem reichhaltigen Angebot:

SpecialFolder	Beschreibung
DesktopDirectory	Verzeichnis für Desktop-Dateien
Fonts	Ein virtueller Ordner mit Schriftarten
LocalApplicationData	Der Ordner für lokale Anwendungsdaten
MyComputer	Der *Arbeitsplatz*-Ordner
MyDocuments	Der Ordner *Eigene Dateien*
MyMusic	Der Ordner *Eigene Musik*
MyPictures	Der Ordner *Eigene Bilder*
MyVideos	Der Ordner *Eigene Videos*
System	Das *System*-Verzeichnis
UserProfile	Der Ordner des aktuellen Benutzers
Windows	Das *Windows*-Verzeichnis

Eine konkrete Anwendung des *FolderBrowserDialog*s finden Sie im Praxisbeispiel

► 7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen

7.7 Praxisbeispiele

7.7.1 Infos über Verzeichnisse und Dateien gewinnen

Dieses Beispiel demonstriert Ihnen nicht nur den Einsatz der *DirectoryInfo*- und *FileInfo*-Klasse, sondern auch weiteres nützliches Handwerkszeug, wie z.B. die sinnvolle Verknüpfung zweier *ListBox*-Komponenten oder das Auswerten der Eingabetaste bei einer *TextBox*.

Oberfläche

Auf *Form1* platzieren Sie eine *TextBox*-, zwei *ListBox*- und zwei große *Label*-Komponenten im 3D-Outfit (siehe Laufzeitabbildung am Beispielende).

Quelltext

```
using System.IO;

public partial class Form1 : Form
{
 ...
```

Globale Deklarationen auf Form-Ebene:

```
    private string myRoot = @"C:\";            // übergeordnetes Verzeichnis
    private string myDirName, myFileName;
    private const char CrLf = (char) 10;      // für Zeilenumbruch in den Labels erforderlich
```

Die Startaktivitäten können in der überschriebenen *OnLoad*-Methode erledigt werden:

```
   protected override void OnLoad(System.EventArgs e)
   {
       textBox1.Text = myRoot;
       showDirectories();
   }
```

Die folgende Methode zeigt alle zu *myRoot* untergeordneten Verzeichnisse in *listBox1* an:

```
   private void showDirectories()
   {
       DirectoryInfo[] myDirectories;         // Array zum Speichern der Unterverzeichnisse
```

Erzeugen eines neuen *DirectoryInfo*-Objekts, welches auf das Rootverzeichnis zeigt:

```
       DirectoryInfo myDirectory = new DirectoryInfo(myRoot);
```

Alle Unterverzeichnisse ermitteln und abspeichern (vorher Anzeige löschen):

```
       listBox1.Items.Clear();
       myDirectories = myDirectory.GetDirectories();
```

Alle Verzeichnisse durchlaufen und Verzeichnisnamen zur *ListBox* hinzufügen:

```
       for (int i = 0; i < myDirectories.Length; i++)
           listBox1.Items.Add(myDirectories[i].Name);
```

Der erste Eintrag in der Verzeichnis-*ListBox* wird selektiert, dadurch wird das *SelectedIndex-Changed*-Event ausgelöst:

```
       listBox1.SelectedIndex = 0;
   }
```

Synchronisieren aller Files in der Datei-*ListBox* mit dem selektierten Verzeichnis und Anzeige der Verzeichnis-Informationen:

```
   private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e)
   {
        DirectoryInfo myDirectory;           // aktuelles Verzeichnis
```

```
        string dirInfo = "";
        FileInfo[] myFiles;                 // Array für alle Dateiinformationen
        listBox2.Items.Clear();             // aktuellen Inhalt löschen
```

DirectoryInfo-Objekt aufgrund des selektierten *ListBox*-Eintrags erzeugen:

```
        myDirName = listBox1.SelectedItem.ToString() + @"\";
        myDirectory = new DirectoryInfo(myRoot + myDirName);
```

Verzeichnis-Infos zusammensetzen:

```
        dirInfo += "Pfad: " + myDirectory.FullName + CrLf;
        dirInfo += "Erstellungsdatum: " + myDirectory.CreationTime + CrLf;
        dirInfo += "Attribute: " + myDirectory.Attributes.ToString() + CrLf;
        label1.Text = dirInfo;
```

Alle im Verzeichnis enthaltenen Dateien dem *FileInfo*-Array zuweisen:

```
        myFiles = myDirectory.GetFiles();
```

File-Array durchlaufen und die Dateien zur *ListBox* hinzufügen:

```
       if (myFiles.Length > 0)
       {
           for (int i = 0; i < myFiles.Length; i++)
                listBox2.Items.Add(myFiles[i].Name);
```

Der erste Eintrag in der Datei-*ListBox* wird selektiert, dadurch wird deren *SelectedIndexChanged*-Event ausgelöst:

```
          listBox2.SelectedIndex = 0;
       }
   }
```

Zweck des folgenden Event-Handlers ist es, den Inhalt des Labels *Dateieigenschaften* zu aktualisieren:

```
   private void listBox2_SelectedIndexChanged(object sender, System.EventArgs e)
   {
       FileInfo myFile; string fileInf = "";
       yFileName = listBox2.SelectedItem.ToString();
```

Neues *File*-Objekt erzeugen:

```
       myFile = new FileInfo(myRoot + myDirName + myFileName);
```

Datei-Infos zusammensetzen und anzeigen:

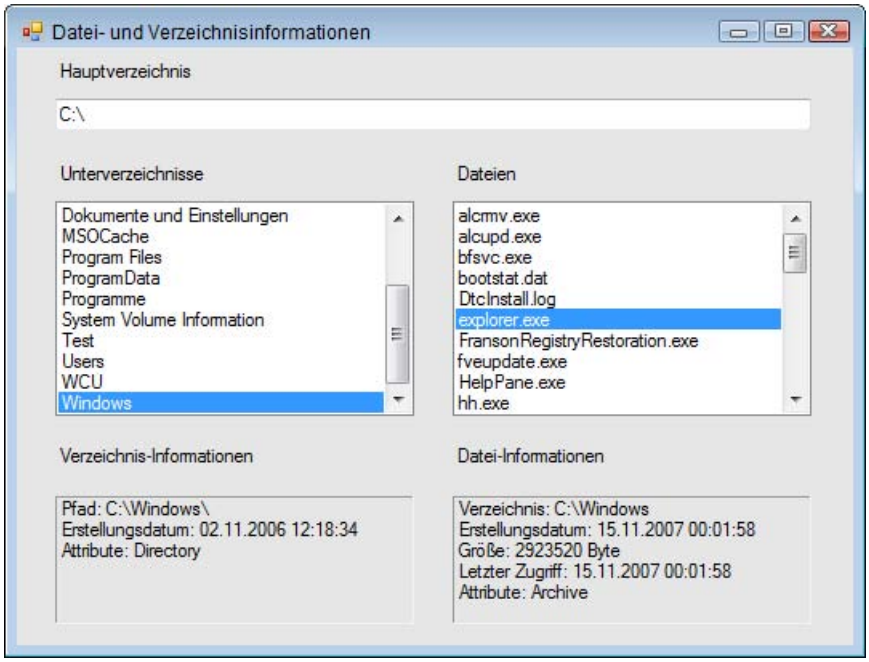
```
       fileInf += "Verzeichnis: " + myFile.DirectoryName + CrLf;
       fileInf += "Erstellungsdatum: " + myFile.CreationTime + CrLf;
       fileInf += "Größe: " + myFile.Length + " Byte" + CrLf;
       fileInf += "Letzter Zugriff: " + myFile.LastAccessTime + CrLf;
       fileInf += "Attribute: " + myFile.Attributes.ToString() + CrLf;
       label2.Text = fileInf;
   }
```

Nachdem Sie das übergeordnete Verzeichnis editiert haben, kann die Eingabetaste ausgewertet werden, um die Eingabe abzuschließen:

```
    private void textBox1_KeyUp(object sender, System.Windows.Forms.KeyEventArgs e)
    {
        if (e.KeyCode == Keys.Enter)
        {
            myRoot=textBox1.Text;
            if (!myRoot.EndsWith("/")) myRoot += "/";
            showDirectories();
    }
...
```

Test

Bei Programmstart erscheinen zunächst in der linken *ListBox* alle Unterverzeichnisse zur Root *C:*. Klicken Sie nun auf ein Unterverzeichnis, um sich in der rechten *ListBox* die darin enthaltenen Dateien anzeigen zu lassen.



Wenn Sie sich in der Verzeichnishierarchie von oben nach unten "hangeln" wollen, müssen Sie das Hauptverzeichnis in der *TextBox* per Hand ergänzen und die Eingabe mittels Entertaste abschließen.

HINWEIS: Um den Code überschaubar zu halten, wurde auf eine Fehlerbehandlung (z.B. bei Eingabe ungültiger Verzeichnisse) verzichtet (siehe Kapitel 11).

7.7.2 Eine Verzeichnisstruktur in die TreeView einlesen

Sie möchten eine ähnliche Funktionalität wie im Windows-Explorer bereitstellen? Nichts ist dazu besser geeignet als das *TreeView*-Control! Das rekursive Durchsuchen des Dateisystems ist allerdings ziemlich zeitaufwändig, sodass es recht lange dauern kann, bis der Verzeichnisbaum vollständig ist.

Das vorliegende Beispiel nutzt das Ereignis *BeforeExpand* der *ListView*, um die benötigten Verzeichnisinformationen zur Laufzeit erst dann zu ermitteln, wenn sie tatsächlich benötigt werden, was allerhand Zeit sparen kann.

Oberfläche

Auf dem Startformular *Form1* platzieren Sie links eine *TreeView* und rechts eine *ListBox* – das genügt!

Quellcode

```
using System.IO;
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
```

Die Vorbereitungen:

```
        TreeNode rootNode = new TreeNode(@"C:\");   // Wurzelknoten erzeugen
        treeView1.Nodes.Add(rootNode);
        addChildNodes(rootNode);                    // untergeordnete Ebene füllen und
        treeView1.Nodes[0].Expand();                // ... expandieren
    }
```

Die Hauptarbeit erledigt die Methode *addChildNodes*, welcher als Parameter ein Knoten (*dirNode*) übergeben wird. Im Ergebnis werden alle Knoten der untergeordneten Verzeichnisebene hinzugefügt:

```
    private void addChildNodes(TreeNode dirNode)
    {
        DirectoryInfo dir = new DirectoryInfo(dirNode.FullPath);
        try
        {
```

Alle Unterverzeichnisse durchlaufen:

```
            foreach (DirectoryInfo dirItem in dir.GetDirectories())
            {
```

Einen Child-Knoten für jedes Unterverzeichnis hinzufügen:

```
                TreeNode newNode = new TreeNode(dirItem.Name);
                dirNode.Nodes.Add(newNode);
```

Jeder Child-Knoten erhält selbst wiederum einen einzelnen Child-Knoten, der mit einem Platzhalterzeichen (*) gekennzeichnet ist:

```
                newNode.Nodes.Add("*");
            }
        }
        catch (UnauthorizedAccessException err)
        {
          MessageBox.Show(err.ToString());
        }
    }
```

Ein Knoten wurde expandiert (aber die untergeordnete Ebene noch nicht gezeichnet):

```
    private void treeView1_BeforeExpand(object sender, TreeViewCancelEventArgs e)
    {
```

Falls es sich beim ersten Child-Knoten um einen Platzhalterknoten handelt, wird dieser gelöscht und die Verzeichnisebene neu erstellt:

```
        if (e.Node.Nodes[0].Text == "*")
        {
            treeView1.BeginUpdate();   // erneutes Zeichnen deaktivieren
            e.Node.Nodes.Clear();      // Platzhalterknoten löschen
            addChildNodes(e.Node);     // alle untergeordneten Knoten hinzufügen
            treeView1.EndUpdate();     // erneutes Zeichnen aktivieren
        }
    }
```

Die Knoten-Auswahl wurde durch den Anwender geändert:

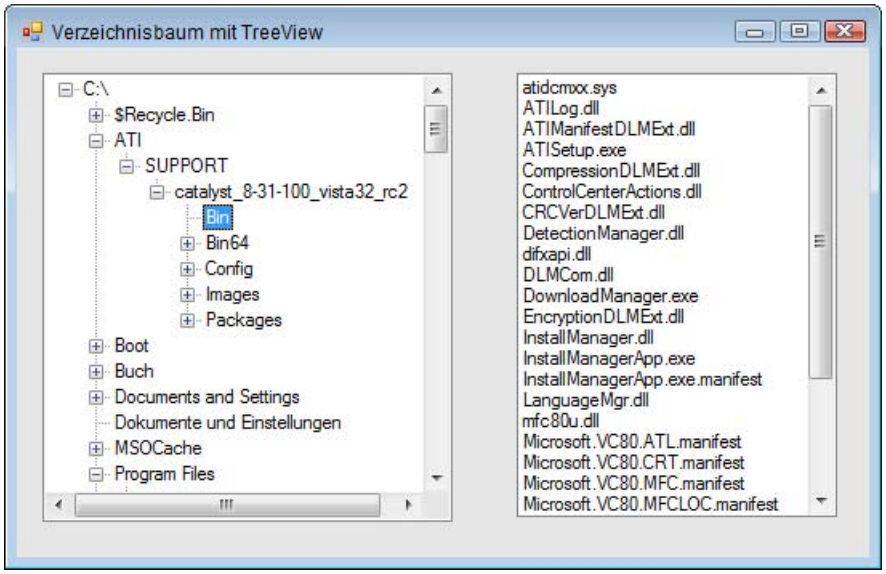
```
    private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
    {
```

Alle im entsprechenden Verzeichnis enthaltenen Dateien werden in der *ListBox* angezeigt:

```
        DirectoryInfo dir = new DirectoryInfo(e.Node.FullPath);
        listBox1.Items.Clear();
        listBox1.Items.AddRange(dir.GetFiles());
    }
}
```

Test

Bewegen Sie sich durch die *TreeView*! Klicken Sie links auf ein bestimmtes Verzeichnis, werden rechts die darin enthaltenen Dateien angezeigt.

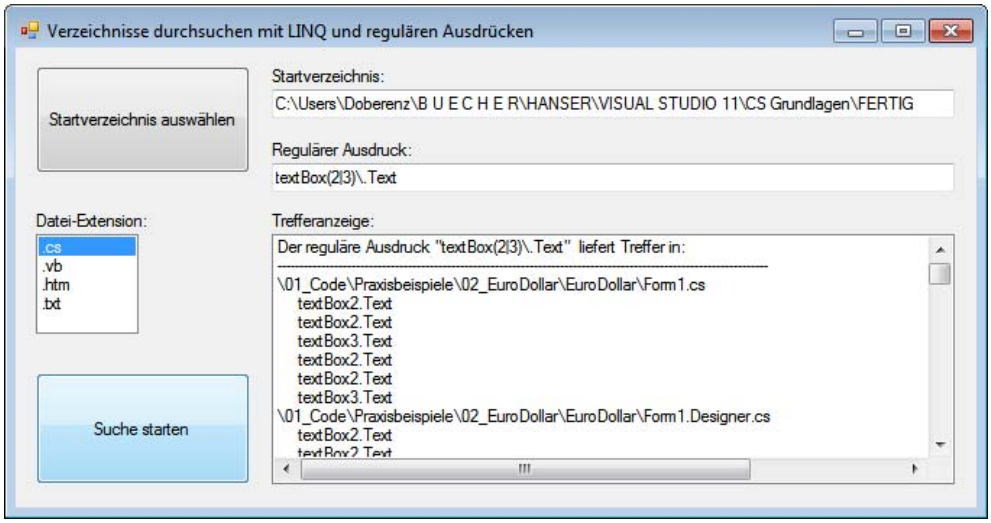


7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen

Mit dem Know-how dieser ausbaufähigen und durchaus praxistauglichen Demo lassen sich große Mengen von Textdokumenten in einem kompletten Verzeichnisbaum nach bestimmten Begriffen und Kriterien durchsuchen. Dabei geht es um den kombinierten Einsatz von Datei-Operationen zusammen mit LINQ-Abfragen, die von regulären Ausdrücken unterstützt werden.

Oberfläche

Auf dem Startformular *Form1* platzieren wir je zwei *Button*s, *TextBox*en und *ListBox*en.



Zur Auswahl des Startverzeichnissen benötigen wir noch einen *FolderBrowserDialog*. *Die Items*-Auflistung der *listBox2* (links) füllen wir im Zeichenfolgen-Editor mit verschiedenen Dateiextensions.

Quellcode

Folgenden Namespaces sind für unser Beispiel von besonderer Bedeutung:

```
...
using System.Linq;
using System.IO;
using System.Text.RegularExpressions;

public partial class Form1 : Form
{
   ...
```

Private Variablen für das Startverzeichnis und den gewählten Datei-Typ:

```
        private string startFolder = "";
        private string ext = "";
```

Das Startverzeichnis auswählen:

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Anfangseinstellung ist Ihr Benutzer-Verzeichnis:

```
            folderBrowserDialog1.RootFolder = System.Environment.SpecialFolder.UserProfile;
            DialogResult result = this.folderBrowserDialog1.ShowDialog();
            if (result == DialogResult.OK)
            {
                startFolder = this.folderBrowserDialog1.SelectedPath;
                textBox4.Text = startFolder;
                button2.Enabled = true;
            }
        }
```

Die folgende (generische) Hilfs-Methode liefert eine Liste aller *FileInfo*-Objekte in allen Unterverzeichnissen des Startverzeichnisses, dabei wird vorausgesetzt, dass die Anwendung über die entsprechenden Zugriffsrechte verfügt:

```
        private IEnumerable<FileInfo> getAllFiles(string path)
        {
```

Überprüfen, ob das Verzeichnis vorhanden ist:

```
            if (!Directory.Exists(path))
                throw new DirectoryNotFoundException();
```

Der Puffer für die Dateinamen:

```
            string[] fileNames = null;
```

Die generische Liste:

```
List<FileInfo> files = new List<FileInfo>();
```

Alle Dateien in allen Unterverzeichnissen werden ermittelt:

```
fileNames = Directory.GetFiles(path, "*" + ext, SearchOption.AllDirectories);
foreach (string name in fileNames)
{
    files.Add(new FileInfo(name));
}
return files;
}
```

Die Schaltfläche "Suche starten":

```
private void button2_Click(object sender, EventArgs e)
{
    listBox1.Items.Clear();
```

Den regulären Ausdruck zuweisen:

```
string rxPattern = textBox1.Text;
Regex rgx = new Regex(rxPattern);
```

Datei-Extension zuweisen:

```
ext = listBox2.SelectedItem.ToString();
```

Alle Dateien in eine generische Liste laden (siehe obige Hilfsmethode):

```
IEnumerable<FileInfo> fileList = getAllFiles(startFolder);
```

Die LINQ-Abfrage erzeugt eine Liste der Dateien, in denen es Treffer gibt, und eine Liste der Treffer in jeder Datei:

```
var qryMatchFiles =
    from file in fileList
```

Eine *where*-Klausel ist eigentlich überflüssig, da die Dateiliste bereits nach Dateityp gefiltert ist:

```
//    where file.Extension == ext
```

ReadAllText öffnet die Textdatei, liest alle Zeilen und schließt die Datei wieder:

```
    let fileText = File.ReadAllText(file.FullName)
```

Jetzt erfolgt in der Datei die Suche nach Treffern unseres regulären Ausdrucks:

```
    let matches = rgx.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
```

Im folgenden LINQ-Ausdruck ist eine explizites Typecasting erforderlich, da *Match* kein Mitglied einer generische *IEnumerable* Collection ist:

```
                matchValues = from Match match in matches
                              select match.Value
            };

        listBox1.Items.Add("Der reguläre Ausdruck " + rxPattern + "  liefert Treffer in:");
```

In zwei verschachtelten Schleifen werden alle Pfade der Fundstellen und alle Treffer durchfahren (dabei wird obige LINQ-Abfrage ausgeführt[1]):

```
        foreach (var m1 in qryMatchFiles)
        {
```

Pfad der Fundstelle (ohne Startverzeichnis):

```
            string s = m1.name.Substring(startFolder.Length);
```

Verzeichnisnamen ausgeben:

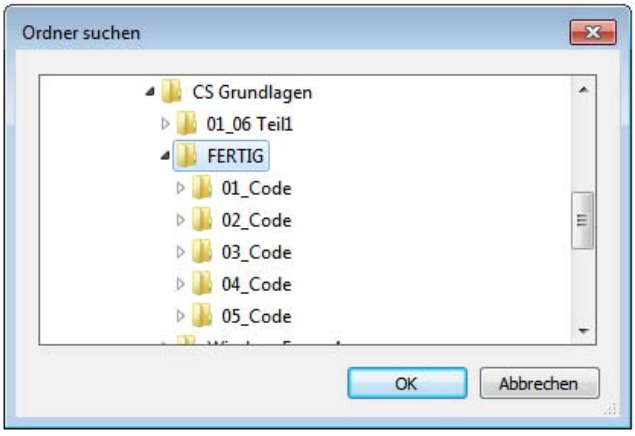
```
            listBox1.Items.Add(s);
```

Alle Treffer in dieser Datei ausgeben:

```
            foreach (var m2 in m1.matchValues)
            {
                listBox1.Items.Add("     " + m2);
            }
        }
    }
}
```

Test

Wählen Sie zunächst den Dateityp und dann das Startverzeichnis aus (das ist die Wurzel des zu durchsuchenden Verzeichnisbaums). Stellen Sie vorher sicher, dass die Zugriffsrechte vorhanden sind und dass alle zu durchsuchenden Dateien geschlossen sind.



[1] LINQ nutzt standardmäßig *deferred query execution* (verzögerte Ausführung einer Abfrage, siehe Kapitel 6).

Jetzt geben Sie Ihren Suchbegriff in Form eines regulären Ausdrucks ein, z.B: *textBox(2|3)\.Text*. Nach Klick auf die Schaltfläche *Suche starten* vergeht mehr oder weniger viel Zeit, bis alle Unterverzeichnisse durchforstet und alle Fundstellen aufgelistet sind. Wie Sie der Laufzeitansicht (siehe Beginn des Beispiels) entnehmen, wurde nach allen Vorkommen von *textBox2.Text* oder *textBox3.Text* in allen (**.cs*) Quellcode-Dateien unterhalb des Startverzeichnisses gesucht.

Bemerkungen

- Wie Sie bereits an diesem einfachen Beispiel sehen können, sind für eine sinnvolle Arbeit mit diesem Programm Grundkenntnisse der Syntax regulärer Ausdrücke unverzichtbar, vor allem die Kenntnis der Metazeichen ( ), |, \ , ., ?, + usw. (siehe Kapitel 4).
- Wenn Sie beispielsweise nach dem String "1+1=2" suchen, müssen Sie dazu den regulären Ausdruck "1\+1=2" verwenden, da das Pluszeichen ein Metazeichen ist und mit dem Backslash "unescaped" werden muss.
- Durch Voranstellen der Zeichen *(?i)* ignorieren Sie bei der Suche die Groß-/Kleinschreibung.

Kapitel 8

Dateien lesen und schreiben

Haben wir uns bislang nur mit Operationen auf Verzeichnisebene beschäftigt, so kommen wir jetzt endlich zur eigentlichen Dateiarbeit.

Zu den wichtigsten Dateitypen gehören:

- Textdateien,
- Binärdatei (Bilddateien etc.),
- sequenzielle Dateien und
- Dateien mit wahlfreiem Zugriff (Random-Access).

HINWEIS: Da .NET keine typisierten bzw. strukturierten Dateien unterstützt, müssen sequenzielle und Random-Access-Dateien durch geeignete Programmiermaßnahmen auf Binärdateien zurückgeführt werden.

8.1 Grundprinzip der Datenpersistenz

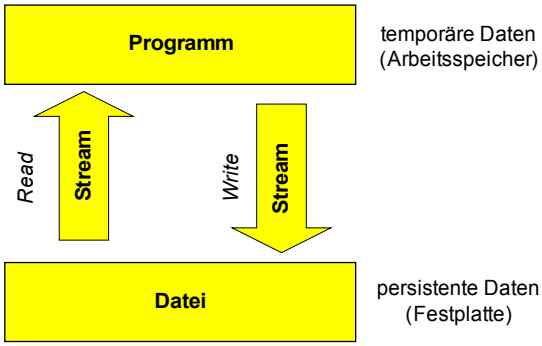
Während ein Programm mit *temporären* bzw. *transienten* Daten arbeitet, die im Arbeitsspeicher abgelegt sind, bezeichnet man die in einer Datei auf Festplatte, CDROM, Stick etc. dauerhaft gespeicherten Informationen als *persistente* Daten.

8.1.1 Dateien und Streams

Wie Sie der folgenden Abbildung entnehmen können, gewährleisten Streams quasi als "Verbindungskanäle" die Kommunikation zwischen Datei und Programm.

Ganz allgemein ist ein Stream eine Aufeinanderfolge von Bytes. Gemeinsame Basisklasse für alle Stream-Klassen ist *System.IO.Stream*. Die wichtigsten davon abgeleiteten Klassen sind *System.IO.FileStream* (Inhalt einer Text- oder Binärdatei), *System.IO.MemoryStream* (Bytefolge im Hauptspeicher), *System.Security.Cryptography.CryptoStream* (verschlüsselte Bytefolge), *System.*-

IO.Compression.GZipStream (komprimierte Bytefolge) oder S*ystem.Net.Sockets.NetworkStream* (Bytefolge, die über ein Netzwerk gesendet wird).



Die Basisklasse *System.IO.Stream* stellt einfache Operationen wie *Read*, *Write* und *Close* sowie Eigenschaften wie *CanRead* und *CanWrite* zur Verfügung. Für komplexere Operationen wie *ReadLine*, *WriteLine*, *Peek* gibt es spezielle Reader-/Writer-Klassenpärchen, z.B. *StreamReader/StreamWriter* für ASCII-Streams oder *BinaryReader/BinaryWriter* für beliebige Bytefolgen.

8.1.2 Die wichtigsten Klassen

Nicht nur zum Kopieren, Löschen und Verschieben von Dateien, sondern auch für das Erzeugen von Stream-Objekten werden in der Regel die Klassen *File* bzw. *FileInfo* benötigt. Die folgende Tabelle zeigt weitere wichtige Klassen für Stream-Operationen.

Klasse	Beschreibung
File	Unterstützt das Erstellen von *FileStream*-Objekten (statisch)
FileInfo	Unterstützt das Erstellen von *FileStream*-Objekten (instanziierbar)
FileStream	Erlaubt, basierend auf einer Datei, das Erstellen einer *Stream*-Instanz
StreamReader	Implementiert ein *TextReader*-Objekt, welches Zeichen von einem Byte-Stream in einer bestimmten Kodierung liest
StreamWriter	Implementiert ein *TextWriter*-Objekt, welches Zeichen in einen Stream in einer speziellen Kodierung liest
StringReader	Implementiert ein *TextReader*-Objekt, das Daten von einem String liest
StringWriter	Implementiert ein *TextWriter*-Objekt, das Daten in einen String schreibt, die Daten werden in einer darunter liegenden *StringBuilder*-Klasse gespeichert
BinaryReader	Erlaubt das binäre Lesen von Dateien
BinaryWriter	Erlaubt das binäre Schreiben in Dateien
BinaryFormatter	Kann Objekte in einen Stream serialisieren bzw. von dort deserialisieren

8.1.3 Erzeugen eines Streams

Voraussetzung für jeglichen Dateizugriff ist das Vorhandensein eines *Stream*-Objekts. Letzteres kann entweder über die *Open*-Methode eines *FileInfo*-Objekts oder der (statischen) *File*-Klasse erzeugt werden.

BEISPIEL 8.1: Die (im Arbeitsverzeichnis befindliche) Datei *temp.txt* soll für den exklusiven Schreib-/ Lesezugriff geöffnet werden. Falls nicht vorhanden, wird sie neu erzeugt.

C#

```
using System.IO;
...
FileInfo myFile = new FileInfo("temp.txt");
FileStream myStream = myFile.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
                                                                    FileShare.None);
```

oder

```
FileStream myStream = File.Open("temp.txt", FileMode.OpenOrCreate, FileAccess.ReadWrite,
                                                                    FileShare.None);
```

Zur Bedeutung der einzelnen Parameter kommen wir im folgenden Abschnitt.

8.2 Dateiparameter

In den Methoden bzw. Konstruktoren der Klassen *File, FileInfo* und *FileStream* werden bestimmte Dateiparameter übergeben, die in Aufzählungstypen (Enumerationen) gekapselt sind.

8.2.1 FileAccess

Diese Enumeration bezeichnet den Zugriffslevel auf eine Datei.

Mitglied	Beschreibung
Read	Erlaubt Lesezugriff
ReadWrite	Erlaubt Lese- und Schreibzugriff
Write	Erlaubt Schreibzugriff

8.2.2 FileMode

Diese Enumeration bestimmt den Öffnungsmodus einer Datei.

Mitglied	Beschreibung
Append	Eine existierende Datei wird geöffnet und der Dateizeiger an das Ende bewegt, oder eine neue Datei wird erstellt (*FileAccess.Write* ist erforderlich, Leseversuche schlagen fehl)

Mitglied	Beschreibung
Create	Eine neue Datei wird erzeugt. Falls die Datei bereits existiert, wird sie überschrieben
Open	Eine existierende Datei wird geöffnet
OpenOrCreate	Falls die Datei existiert, wird sie geöffnet, andernfalls wird sie neu erzeugt
Truncate	Eine existierende Datei wird geöffnet und die Dateigröße auf null Bytes beschnitten

8.2.3 FileShare

Diese Enumeration verwenden Sie, um festzulegen, ob auf eine Datei gleichzeitig von mehreren Prozessen aus zugegriffen werden kann.

Mitglied	Beschreibung
None	Die Datei ist für den gleichzeitigen Zugriff gesperrt. Alle weiteren Anforderungen zum Öffnen werden abgelehnt, es sei denn, die Datei ist geschlossen
Read	Auch andere Benutzer bzw. Prozesse dürfen die Datei lesen. Versuche zum Schreiben bzw. Abspeichern schlagen fehl
ReadWrite	Die Datei kann von mehreren Benutzern bzw. Prozessen sowohl zum Lesen als auch zum Schreiben geöffnet werden (problematisch, da der letzte Benutzer auch die Änderungen anderer Benutzer abspeichert)
Write	Die Datei ist für den gleichzeitigen Schreibzugriff geöffnet. In Kombination mit Read kann das den *ReadWrite*-Parameter ersetzen

BEISPIEL 8.2: Mittels *FileStream*-Konstruktor wird eine vorhandene Datei geöffnet und weiteren Benutzern der schreibgeschützte Zugriff gewährt (*FileShare.Read*).

C#
```
using System.IO;
...
FileStream myStream = new FileStream(@"c:\test.txt", FileMode.Open,
                                     FileAccess.Read,FileShare.Read);
```

8.3 Textdateien

Das Lesen und Schreiben von Textdateien gestaltet sich im .NET-Framework mit den Klassen *StreamReader* und *StreamWriter* (beide von *TextReader*/*TextWriter* abgeleitet) ziemlich einfach.

8.3.1 Eine Textdatei beschreiben bzw. neu anlegen

Um eine neue Textdatei anzulegen, verwenden Sie die Methode *CreateText* der *File*- bzw. *FileInfo*-Klasse.

BEISPIEL 8.3: Textdatei erzeugen und mit dem Inhalt einer *TextBox* füllen.

C#
```
using System.IO;
...
StreamWriter writer = File.CreateText("Liesmich.txt");
foreach (string s in textBox1.Lines)  writer.WriteLine(s);
writer.Close();
```

Die *AppendText*-Methode erstellt einen *StreamWriter* zum Hinzufügen von Text zu einer Textdatei. Falls die Datei nicht vorhanden ist, wird sie neu erzeugt.

BEISPIEL 8.4: Zehnmal "Hallo" an das Ende einer vorhandenen Textdatei schreiben.

C#
```
using System.IO;
...
string pfad =  "Liesmich.txt";
FileInfo file = new FileInfo(pfad);
StreamWriter writer = file.AppendText();
for (int i = 0; i < 10; i++)
{
    writer.Write("Hallo");
    writer.Write(Environment.NewLine);
}
writer.Close();
```

WriteAllLines/AppendAllText

Die *WriteAllLines*-Methode erstellt eine neue Datei (ist die Zieldatei bereits vorhanden, wird sie überschrieben). Die in einem Array übergebene Zeichenfolge wird in die Datei geschrieben, danach wird die Datei geschlossen.

Ähnlich verhält sich die *AppendAllText*-Methode, sie öffnet eine Datei, um eine übergebene Zeichenfolge an deren Ende hinzuzufügen und schließt die Datei wieder.

BEISPIEL 8.5: *WriteAllLines/AppendAllText*

C#

Wenn die Datei nicht vorhanden ist, wird sie angelegt, mit drei Textzeilen gefüllt und wieder geschlossen. Anschließend wird sie nochmals geöffnet, um eine weitere Zeile anzufügen.

```
string pfad = "Liesmich.txt";
if (!File.Exists(pfad))
{
   string[] text = { "Liebe C#-Freunde", "dies hier ist eine Textdatei", "zum Testen " };
   File.WriteAllLines(pfad, text);
}
```

Datei nochmals öffnen, um eine weitere Textzeile hinzuzufügen:

```
string text2 = "... und hier wird noch etwas" + Environment.NewLine +
                                                "an das Ende der Datei angehängt!";
File.AppendAllText(pfad, text2);
```

8.3.2 Eine Textdatei lesen

Sowohl die *File*- als auch die *FileInfo*-Klasse bieten eine Methode *OpenText*, die das Öffnen einer Textdatei und das Auslesen mit Hilfe der zurückgegebenen *StreamReader*-Instanz ermöglicht.

BEISPIEL 8.6: Der Inhalt einer Textdatei wird in einer *TextBox* angezeigt.

C#
```
using System.IO;
...
string output = "";
StreamReader reader = File.OpenText("Liesmich.txt");
while((output = reader.ReadLine()) != null) textBox1.Text += output + Environment.NewLine;
reader.Close();
```

Die *ReadToEnd*-Methode vereinfacht den Leseprozess.

BEISPIEL 8.7: Eine alternative Variante des Vorgängerbeispiels

C#
```
...
StreamReader reader = new StreamReader("Liesmich.txt");
textBox1.Text += reader.ReadToEnd();
reader.Close();
```

Noch einfacher geht es allerdings mit der *ReadAllText*-Methode der *File*-Klasse, die den *StreamReader* überflüssig macht:

BEISPIEL 8.8: Die kürzeste Variante des Vorgängerbeispiels

C#
```
...
textBox1.Text = File.ReadAllText("Liesmich.txt");
```

ReadAllLines/ReadLines

ReadAllLines liest alle Zeilen der Testdatei in ein String-Array ein und schließt dann die Datei.

BEISPIEL 8.9: Auslesen einer Textdatei und Anzeige des Inhalts in einer *ListBox*

C#
```
string[] lines = File.ReadAllLines("Liesmich.txt");
foreach (var line in lines)
    listBox1.Items.Add(line);
```

Doch die Sache hat einen Haken: Bevor die Methode *ReadAllLines* ihren Rückgabewert abliefern kann, muss sie **alle** Zeilen gelesen und ein Array für alle Werte reserviert und gefüllt haben. Das kann bei großen Textdateien zu Zeit- und Speicherplatzproblemen führen[1].

Eine Lösung wäre der Einsatz eines *TextReader* um die Datei zu öffnen und diese zeilenweise in den Speicher einzulesen.

[1] Denken Sie beispielsweise an ein Telefonbuch mit Millionen von Einträgen!

BEISPIEL 8.10: Alternative Lösung des Vorgängerbeispiels, die für größere Textdateien geeigneter ist

C#
```
using (TextReader reader = new StreamReader("Liesmich.txt"))
{
    string line;
    while ((line = reader.ReadLine()) != null)
        listBox1.Items.Add(line);
}
```

HINWEIS: Da obiger Code etwas umständlich ist, wurde ab .NET 4.0 die Methode *ReadLines* eingeführt, die das Gleiche leistet, aber übersichtlicher zu programmieren ist.

ReadLines liefert einen *IEnumerable<T>* anstatt eines Arrays und man muss nicht mehr warten bis alle Zeilen in den Speicher gelesen wurden, bevor man über die Zeilen iterieren kann. Der Aufruf von *ReadLines* kommt sofort zurück!

BEISPIEL 8.11: Eine einfachere Lösung des Vorgängerbeispiels

C#
```
IEnumerable<string> lines = File.ReadLines(pfad);
foreach (var line in lines)
    listBox1.Items.Add(line);
```

Die Methode *File.ReadLines* hat noch einen weiteren Vorteil: man kann die Schleife zu einem beliebigen Zeitpunkt wieder verlassen, ohne wertvolle Zeit durch das Einlesen zusätzlicher Zeilen zu verlieren.

BEISPIEL 8.12: Abbruch des Lesevorgangs wenn die Zeile mehr als 30 Zeichen hat

C#
```
IEnumerable<string> lines = File.ReadLines(pfad);
foreach (var line in lines)
{
    listBox1.Items.Add(line);
    if (line.Length > 30) break;
}
```

8.4 Binärdateien

Der Zugriff auf binäre Daten ist dem auf die zuvor beschriebenen Textdateien sehr ähnlich.

8.4.1 Lese-/Schreibzugriff

Auf Basis einer Instanz der *FileStream*-Klasse können die Klassen *BinaryReader* und *BinaryWriter* zum Lese-/Schreibzugriff instanziiert werden.

Zum Auslesen des jeweils nächsten Zeichens stellt Ihnen *BinaryReader* vielfältige Methoden zur Verfügung (*ReadBoolean*, *ReadByte, ReadInt32*, *ReadDouble*, *ReadChar*, *ReadString* ...).

HINWEIS: Über die Methode *PeekChar* kann das Dateiende abgeprüft werden (liefert dann –1).

Der Schreibzugriff ist über eine der zahlreichen Überladungen der *Write*-Methode des *BinaryWriter* möglich.

BEISPIEL 8.13: Byteweises Auslesen eines Bildes und Kopieren in ein anderes[1].

C#
```
FileStream file_A = new FileStream("c:\Bild_A.bmp", FileMode.Open);
BinaryReader reader = new BinaryReader(file_A);
FileStream file_B = new FileStream("c:\Bild_B.bmp", FileMode.OpenOrCreate);
BinaryWriter writer = new BinaryWriter(file_B);
while (reader.PeekChar() > -1)
            writer.Write(reader.ReadByte());   // liest und schreibt ein Byte
writer.Close();
reader.Close();
```

8.4.2 Die Methoden ReadAllBytes und WriteAllBytes

Mit Einführung der *ReadAllBytes/WriteAllBytes*-Methodenpärchens wurde seit .NET 2.0 auch hier den Zugriff vereinfacht:

BEISPIEL 8.14: Alternative Realisierung des Vorgängerbeispiels (der Dateiinhalt wird in einem *Byte*-Array zwischengespeichert).

C#
```
byte[] bytes = File.ReadAllBytes("c:\Bild_A.bmp");    // Datei => Byte-Array
File.WriteAllBytes("c:\Bild_B.bmp", bytes);          // Byte-Array => Datei
```

8.4.3 Erzeugen von BinaryReader/BinaryWriter

Der Konstruktor von *BinaryReader*/*BinaryWriter* erwartet als Argument ein *FileStream*-Objekt. Es bleibt Ihnen überlassen, ob Sie dieses direkt erzeugen oder aber indirekt über die *OpenRead/OpenWrite*-Methode eines *FileInfo*-Objekts.

BEISPIEL 8.15: Zwei Varianten zum Erzeugen eines *BinaryReader*-Objekts

C#

Variante 1:

```
FileStream readStream = new FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Read);
BinaryReader reader = new BinaryReader(readStream);
```

Variante 2:

```
FileInfo file As New FileInfo(pfad);
BinaryReader reader = new BinaryReader(file.OpenRead());
```

[1] Eine komplette Datei lässt sich natürlich auch durch Anwenden der (statischen) *Copy*-Methode der *File*-Klasse kopieren.

8.5 Sequenzielle Dateien

Sequenzielle Dateien sind gewöhnlich auch typisiert, d h., sie enthalten gleichartig aufgebaute Datensätze. Ein wahlfreier Zugriff (Random Access) ist in der Regel nicht möglich, um z.B. einen bestimmten Datensatz zu erreichen, müssen zunächst die davor stehenden Datensätze hintereinander ausgelesen werden.

8.5.1 Lesen und schreiben von strukturierten Daten

Da .NET typisierte Dateien nicht direkt unterstützt, müssen Sie sich selbst um deren Strukturierung kümmern, Sie müssen also typisierte Dateien quasi wie Binärdateien behandeln.

BEISPIEL 8.16: Lesen und schreiben von strukturierten Daten

C#

Ein Array *pListe*, welches die Daten eines *struct*-Datentyps (*Nachname*, *Geburtsdatum*, *Student (Ja/Nein)*) speichert, wird in einer sequenziellen Datei zwischengespeichert. Diese Methode schreibt den Arrayinhalt in die Datei:

```
private void writeFile(string pfad)
{
  FileStream wStream = new FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Write);
  BinaryWriter binWriter = new BinaryWriter(wStream);
  for (int i=0; i< pmax; i++)
  {
```

Die *Write*-Methode des *BinaryWriter* verfügt über eine Überladung für fast jeden Datentyp:

```
    binWriter.Write(pListe[i].vorName);
    binWriter.Write(pListe[i].nachName);
    binWriter.Write(pListe[i].geburt.ToShortDateString());
    binWriter.Write(pListe[i].student);
  }
  binWriter.Flush();                        // Puffer => Disk
  binWriter.Close();
  wStream.Close();
}
```

Die folgende Methode macht es umgekehrt, sie füllt ein Array mit dem Dateinhalt:

```
private void readFile(string pfad)
{
  FileStream rStream = new FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Read);
  BinaryReader binReader = new BinaryReader(rStream);
  if (rStream.Length > 0)                   // nicht bei neu erzeugter Datei!
  {
    for (int i=0; i< pmax; i++)
    {
```

Der *BinaryReader* verfügt für fast jeden Datentyp über eine spezielle Lesemethode:

```
      pListe[i].vorName = binReader.ReadString();
```

BEISPIEL 8.16: Lesen und schreiben von strukturierten Daten

```
        pListe[i].nachName = binReader.ReadString();
        pListe[i].geburt = Convert.ToDateTime(binReader.ReadString());
        pListe[i].student = binReader.ReadBoolean();
      }
    }
    binReader.Close();
    rStream.Close();
  }
```

8.5.2 Serialisieren von Objekten

Nicht nur einfache und strukturierte Datentypen, auch komplette Objekte können in eine Datei geschrieben bzw. von dort gelesen werden. Voraussetzung ist eine Serialisierung, wie sie durch Voranstellen des *Serializable*-Attributs vor die Klassendeklaration vorbereitet wird. Eine zentrale Rolle spielt dabei das *BinaryFormatter*-Objekt, welches hier quasi *BinaryReader* und *BinaryWriter* ersetzt.

BEISPIEL 8.17: Ein *CKunde*-Objekt in eine sequenzielle Datei schreiben und anschließend wieder lesen.

C#

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
...
[Serializable]    // Wichtig!
public class CKunde
     ...          // irgendeine Klassenimplementierung
```

Objekt serialisieren und in Datei schreiben:

```
CKunde kunde = new CKunde(10, Max, Müller, 100.65);   // initialisiertes Record-Objekt
erzeugen
FileStream fileOutput = new FileStream(@"c:\Kunden.dat", FileMode.OpenOrCreate,
FileAccess.Write);
BinaryFormatter binFttr = new BinaryFormatter();
binFttr.Serialize(fileOutput, kunde);
```

Objekt aus Datei zurück lesen und wieder "zusammenbauen" (deserialisieren):

```
CKunde kunde = new CKunde();                          // leeres Record-Objekt erzeugen
FileStream fileInput = new FileStream(@"c:\Kunden.dat", FileMode.Open, FileAccess.Read);
kunde = (CKunde) binFttr.Deserialize(fileInput));     // explizite Typumwandlung
erforderlich!
```

Das Serialisieren/Deserialisieren von Daten auch als Marshalling/Unmarshalling bekannt – ermöglicht die Datenübertragung über Prozessgrenzen hinweg.

Die drei .NET-Serialisierer (Binär, SOAP, XML) unterstützen nicht nur die Serialisierung einzelner Objekte, sondern auch die kompletter Objektbäume (siehe Praxisbeispiel 8.8.2).

8.6 Dateien verschlüsseln und komprimieren

Das einfache Verschlüsseln von Dateien wird z.B. den Windows-Nutzern bekannt sein. Der Benutzer, der die Datei verschlüsselt hat, merkt davon allerdings nichts und muss kein Kennwort eingeben um den Inhalt zu sehen. Will aber ein anderer Benutzer die Datei lesen, kommt eine Fehlermeldung, und der Zugriff auf die Datei wird verweigert.

HINWEIS: Wenn man die Dateiverschlüsselung nutzt, sollte man auf jeden Fall das so genannte "Zertifikat zur Datenverschlüsselung" sichern, da es ohne dieses nicht möglich ist nach einem Systemcrash mit folgender Neuinstallation an die verschlüsselten Daten heranzukommen.

8.6.1 Das Methodenpärchen Encrypt/Decrypt

Die *File*-Klasse verfügt über die Methoden *Encrypt* und *Decrypt*, mit denen eine Datei auf einfache Weise verschlüsselt und entschlüsselt werden kann.

BEISPIEL 8.18: Die Datei *Beispiel.txt* wird ver- und entschlüsselt

C#
```
using System.IO;
...
File.Encrypt("Beispiel.txt");
File.Decrypt("Beispiel.txt");
```

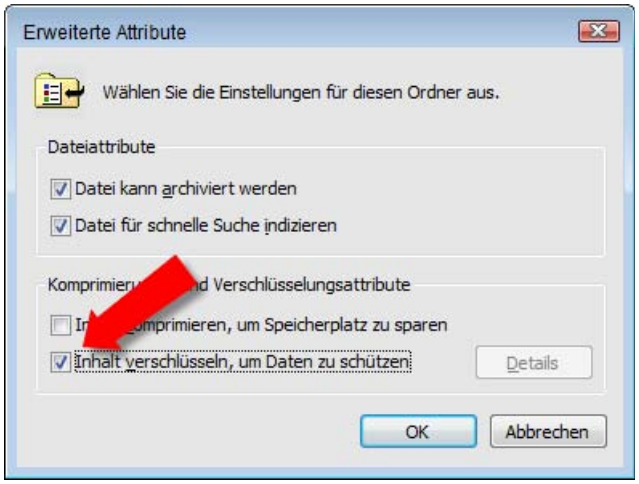
HINWEIS: Erkennen kann man die Verschlüsselung durch die grüne Farbe des Dateinamens und – wenn die Dateiattribute angezeigt werden – durch das E-Attribut (für *Encrypted*).

8.6.2 Verschlüsseln unter Vista/Windows 7/Windows 8

Um eine Datei oder einen Ordner zu verschlüsseln wählen Sie im Kontextmenü den Eintrag *Eigenschaften* und klicken dann auf die Schaltfläche *Erweitert...*. Es öffnet sich das Dialogfenster *Erweiterte Attribute*, in welchem Sie das Häkchen bei *Inhalt verschlüsseln, um Daten zu schützen* setzen (siehe folgende Abbildung). Um die Verschlüsselung wieder aufzuheben, gehen Sie den umgekehrten Weg.

Wie aber können Sie Ihre verschlüsselten Dateien anderen Anwendern zugänglich machen?

Klicken Sie dazu auf die Schaltfläche *Details* im Fenster *Erweiterte Attribute*. Im Fenster *Verschlüsselungsdetails* können Sie sich über die *Hinzufügen*-Schaltfläche die Benutzer anzeigen lassen, für die auf dem Rechner ein Benutzerkonto angelegt ist und die schon mindestens eine Datei verschlüsselt haben. Selektieren Sie den gewünschten User mit der Maus und klicken Sie die *OK*-Schaltfläche.



HINWEIS: Kopiert man eine verschlüsselte Datei auf ein anderes Laufwerk, bleibt die Verschlüsselung nur erhalten, wenn auch dieses mit NTFS formatiert ist, andernfalls entschlüsselt Windows die Datei.

Auf eine andere Art von Verschlüsselung kommen wir im folgenden Abschnitt zu sprechen.

8.6.3 Verschlüsseln mit der CryptoStream-Klasse

Die *CryptoStream*-Klasse im *System.Security.Cryptography*-Namespace ermöglicht das Verschlüsseln und Entschlüsseln des Inhalts eines beliebigen Datenstreams.

Der Konstruktor ist wie folgt definiert:

SYNTAX: `CryptoStream (Stream argument, ICryptoTransform transform, CryptoStreamMode mode)`

Zu den einzelnen Parametern:

- *argument*
  ... definiert den Stream, auf welchem die kryptografische Transformation ausgeführt wird, das kann ein beliebiger von *System.IO.Stream* abgeleiteter Stream sein, beispielsweise eine Instanz von *System.IO.FileStream,* um eine Datei zu verschlüsseln.
- *transform*
  ... definiert die kryptografische Transformation, die auf dem Stream ausgeführt wird. Alle symmetrischen Encryption-/Decryption-Algorithmen, die von der *SymmetricAlgorithm*-Klasse abgeleitet sind, besitzen *CreateEncryptor*- und *CreateDecryptor*-Methoden, die eine Instanz einer *ICryptoTransform*-Implementierung zurückgeben.
- *mode*
  ... definiert, ob Sie auf dem Stream lesend oder schreibend zugreifen wollen, übergeben Sie also entweder die Konstante *CryptoStreamMode.Write* oder *CryptoStreamMode.Read*.

Die *CryptoStream*-Klasse implementiert die standardmäßigen Methoden, um ein Byte-Array vom Stream zu lesen oder ein Byte-Array in den Stream zu schreiben. Der Anwendungscode muss lediglich den Byte-Puffer bereitstellen und die entsprechende Read- oder Write-Methode auf dem Stream aufrufen.

BEISPIEL 8.19: Der Inhalt eines Byte-Arrays wird nach dem Data Encryption Standard (DES) verschlüsselt und in der Datei *EncryptedFile.txt* abgelegt.

C#
```
using System.IO;
using System.Security.Cryptography;
...
private DESCryptoServiceProvider des = new DESCryptoServiceProvider();
ICryptoTransform desEncrypt = des.CreateEncryptor();
FileStream fs = new FileStream("EncryptedFile.txt", FileMode.Create, FileAccess.Write);
CryptoStream cryptoStrm = new CryptoStream(fs, desEncrypt, CryptoStreamMode.Write);
cryptoStrm.Write(byteArr, 0, byteArr.Length);
cryptoStrm.Close();
```

8.6.4 Dateien komprimieren

Mit den Klassen des *System.IO.Compression*-Namespace ist es möglich, Daten zu komprimieren und zu dekomprimieren.

BEISPIEL 8.20: Unter Verwendung der *GZipStream*-Klasse wird der Inhalt einer Datei gepackt und entpackt[1].

C#
```
...
using System.IO;
using System.IO.Compression;

public partial class Form1 : Form
{ ...
    private string QuellDatei = "Test1.txt";
    private string KompDatei = "Test.zip";
    private string ZielDatei = "Test2.txt";

    private byte[] fileBytes = null;   // Puffer
```

Quelldatei lesen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        FileStream strm1 = new FileStream(QuellDatei, FileMode.Open);
        fileBytes = new byte[strm1.Length];
        strm1.Read(fileBytes, 0, fileBytes.Length);
        strm1.Close();
    }
```

[1] Seit .NET 4.0 eignet sich die *GZipStream* Klasse auch zum Komprimieren von Dateien größer 4 GB!

BEISPIEL 8.20: Unter Verwendung der *GZipStream*-Klasse wird der Inhalt einer Datei gepackt und entpackt.

Komprimierte Datei erstellen:

```
private void button2_Click(object sender, EventArgs e)
{
    FileStream strm2 = new FileStream(KompDatei, FileMode.Create);
    GZipStream compStrm = new GZipStream(strm2, CompressionMode.Compress);
    compStrm.Write(fileBytes, 0, fileBytes.Length);
    compStrm.Flush();    // internen Puffer leeren
    compStrm.Close();
    strm2.Close();
}
```

Datei dekomprimieren:

```
private void button3_Click(object sender, EventArgs e)
{
    FileStream strm3 = new FileStream(KompDatei, FileMode.Open);
    GZipStream decompStrm = new GZipStream(strm3, CompressionMode.Decompress);
    StreamReader reader = new StreamReader((Stream)decompStrm);
    File.WriteAllText(ZielDatei, reader.ReadToEnd());
    reader.Close();
    strm3.Close();
}
}
```

8.7 Memory Mapped Files

Was früher nur mit umständlichen API-Aufrufen möglich war, ist seit .NET 4.0 auch direkt per managed Code möglich, dazu bietet der Namespace *System.IO.MemoryMappedFiles* einige spezielle Klassen, welche die Arbeit mit Memory Mapped Files vereinfachen.

Doch was ist eigentlich ein Memory Mapped File (MMF) und wofür können Sie dieses verwenden?

8.7.1 Grundprinzip

Memory Mapped Files (kurz MMF) stellen eine Möglichkeit dar, physische Dateien in den Adressraum einer oder auch mehrerer Anwendungen einzublenden.

Wozu das Ganze? Die Antwort findet sich im Zugriffsprinzip auf derartige "Dateien". Sie verwenden statt umständlicher *Read*- und *Write*-Anweisungen einfach ganz normale Speicheroperationen.

HINWEIS: Obwohl Sie bei MMFs in einem "Speicherbereich" arbeiten, befindet sich die Datei prinzipiell auf der Festplatte, nur der aktuell bearbeitete Abschnitt liegt im Speicher.

Ein weiterer Pluspunkt für die Verwendung von MMFs resultiert aus der strikten Trennung der Anwendungsadressräume. MMFs stellen eine der wenigen Möglichkeiten dar, auf große Datenmengen gleichzeitig von unterschiedlichen Anwendungen aus zuzugreifen, d.h., das MMF wird auch für die Kommunikation zwischen verschiedenen Prozessen (IPC)[1] genutzt.

Ein MMF erlaubt es Ihnen, einen Adressenbereich zu reservieren und physikalischen Speicher zur Verfügung zu stellen. Das hat zumindest zwei Vorteile:

- Es ist günstig, auf Dateien zuzugreifen, die sich auf der Disk befinden, ohne dazu I/O-Operationen durchführen und den Dateiinhalt puffern zu müssen. Das ist besonders bei sehr großen Dateien vorteilhaft.
- Sie können MMFs verwenden um es multiplen Prozessen zu ermöglichen, auf der gleichen Maschine zu laufen und gemeinsam auf dieselben Daten zuzugreifen.

HINWEIS: Ein MMF ist der effektivste Weg für multiple Prozesse, um auf einer einzigen Maschine miteinander zu kommunizieren!

8.7.2 Erzeugen eines MMF

Dazu verwenden Sie verschiedene Methoden der statischen *MemoryMappedFile*-Klasse. Ein MMF wird immer mit einem Namen assoziiert.

CreateFromFile-Methode

Die verschiedenen Überladungen der *CreateFromFile* Methode erzeugen ein MMF von einem spezifischen Pfad oder Filestream einer existierenden Datei auf der Disk. Die Änderungen werden automatisch auf die Disk übertragen, wenn das Mapping aufgehoben wird.

BEISPIEL 8.21: Ein MMF von der Datei *Bild1.bmp* generieren

C#
```
using System.IO.MemoryMappedFiles;    // neu ab .NET 4.0
using System.IO;
...
MemoryMappedFile mmf = MemoryMappedFile.CreateFromFile("Bild1.bmp", FileMode.Open, "MMF1");
```

Wir haben einen der einfachsten Konstruktoren benutzt, um die MMF-Datei *MMF1* zu erzeugen.

Doch das Beste an den MMFs aber ist ihre eingebaute Datenpersistenz:

HINWEIS: Sobald das MMF geschlossen wird, wird der Inhalt auf der Disk gesichert. Das ist ideal, wenn verschiedene Anwendungen auf Informationen zugreifen wollen.

[1] IPC = Interprocess Communication

CreateNew-Methode

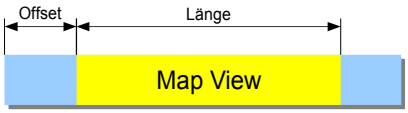
Mit *CreateNew* erzeugen Sie ein MMF das nicht auf eine existierende Datei gemappt ist.

BEISPIEL 8.22: Erzeugen eines MMF mit der maximalen Kapazität von 10000 Bytes

```
C#
long cap = 10000;
MemoryMappedFile mmf = MemoryMappedFile.CreateNew("MMF2", cap));
```

8.7.3 Erstellen eines Map View

Für den Zugriff auf das MMF brauchen Sie einen so genannten Map View (statische Klasse *MemoryMappedViewAccessor*), den Sie mit Hilfe der Methode *CreateViewAccessor* erzeugen. Sie können mehrere Views für ein MMF erstellen oder auch Views auf Teilbereiche der Datei.



BEISPIEL 8.23: Ein Map View für ein MMF (siehe obiges Beispiel) soll den Speicherbereich zwischen den Adressen 1 MByte und 11 MByte (Offset plus Länge) abdecken.

```
C#
long offset = 0x100000;     //  1 MB
long länge = 0xA00000;      // 10 MB
MemoryMappedViewAccessor vacc = mmf.CreateViewAccessor(offset, länge);
```

Wollen Sie Informationen lesen oder schreiben, dann rufen Sie einfach die Map View-Methoden mit dem korrekten Offset auf:

BEISPIEL 8.24: Schreiben einer Zahl in das erste Byte

```
C#
int nr = 12345;
vacc.Write(0, nr);
```

BEISPIEL 8.25: Schreiben eines benutzerdefinierten Typs (generischen Version)

```
C#
vacc.Write<Container>(4, ref MyContainer);
```

BEISPIEL 8.26: Lesezugriff auf das i-te *Byte*

```
C#
vacc.Read(i, out color);
```

Es ist möglich, den gleichen Teil einer Datei an mehrere Adressbereiche zu mappen (konkurrierender Speicher). Damit zwei Views konkurrierend sind, müssen sie vom selben MMF erzeugt werden. Hingegen fügt das Erzeugen von zwei MMFs von derselben Datei zu keiner Konkurrenz. Mehr dazu siehe auch

► 8.8.3 Ein Memory Mapped File (MMF) verwenden

8.8 Praxisbeispiele

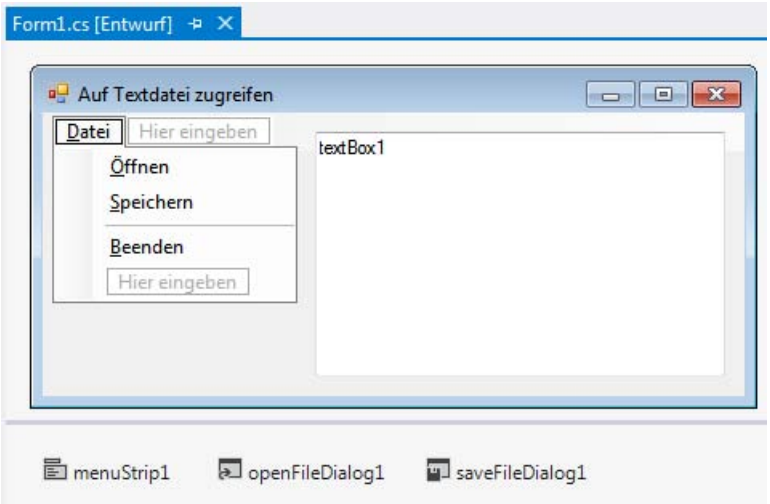
8.8.1 Auf eine Textdatei zugreifen

Das vorliegende Beispiel demonstriert das Öffnen und Speichern einer einfachen Textdatei in Zusammenarbeit mit einer Menükomponente und den Dateidialogen.

Oberfläche

Ziehen Sie vom Werkzeugkasten eine *TextBox* (*MultiLine=True*), eine *MenuStrip*-, eine *OpenFileDialog*- und eine *SaveFileDialog*-Komponente auf das Startformular *Form1*.

Das Erstellen des Hauptmenüs ist nach Doppelklick auf die Menükomponente sehr intuitiv und braucht deshalb hier nicht extra erklärt zu werden:



Quellcode

```
...
using System.IO;

public partial class Form1 : Form
{
    ...
```

Der Dateipfad, um den sich letztendlich alles dreht, wird zweckmäßiger Weise auf Klassenebene deklariert:

```
    private string pfad;
```

Die zahlreichen Anfangseigenschaften der beiden Dateidialoge legen wir beim Laden von *Form1* fest. Die meisten Eigenschaften gelten sowohl für *OpenFileDialog* als auch für *SaveFileDialog*, weshalb wir sie nur einmal zu erklären brauchen:

```
    private void Form1_Load(object sender, EventArgs e)
    {
```

Zunächst der Öffnen-Dialog, wir beginnen mit dem Festlegen der Standard-Dateinamenserweiterung:

```
        openFileDialog1.DefaultExt = "txt";
```

Die Filterzeichenfolge:

```
        openFileDialog1.Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*";
```

Warnung, wenn der Namen einer nicht vorhandenen Datei eingegeben wird:

```
        openFileDialog1.CheckFileExists = true;
```

Das Anfangsverzeichnis:

```
        openFileDialog1.InitialDirectory = Application.ExecutablePath;
```

Die Beschriftung der Titelleiste des Dialogs:

```
        openFileDialog1.Title = "Bitte öffnen Sie eine Textdatei!";
```

Nun zum Speichern-Dialog:

```
        saveFileDialog1.DefaultExt = "txt";
```

Der standardmäßig eingetragene Dateiname:

```
        saveFileDialog1.FileName = "Beispiel.txt";
```

Automatisches Anhängen der *DefaultExt*, falls diese weggelassen wird:

```
        saveFileDialog1.AddExtension = true;
```

Warnung, wenn bereits eine gleichnamige Datei vorhanden ist:

```
        saveFileDialog1.OverwritePrompt = true;
```

Überprüfen, ob Dateiname erlaubte Zeichen enthält:

```
        saveFileDialog1.ValidateNames = true;
```

Weitere Einstellungen:

```
        saveFileDialog1.Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*";
        saveFileDialog1.InitialDirectory = Application.ExecutablePath;
        saveFileDialog1.Title = "Bitte speichern Sie die Textdatei!";
    }
```

Den Rahmencode für die *Click*-Eventhandler der einzelnen Menüeinträge erzeugen wir am einfachsten durch Doppelklick auf den entsprechenden Eintrag:

Der Menüpunkt *Datei|Öffnen*:

```
    private void öffnenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
```

```
            pfad = openFileDialog1.FileName;
            textBox1.Text = String.Empty;
            try
            {
                textBox1.Text = File.ReadAllText(pfad);
            }
            catch
            { }
```

Dateipfad in der Titelleiste des Formulars anzeigen:

```
            this.Text = pfad;
        }
    }
```

Der Menüpunkt *Datei|Speichern*:

```
    private void speichernToolStripMenuItem_Click(object sender, EventArgs e)
    {
        if (saveFileDialog1.ShowDialog() == DialogResult.OK)
        {
            pfad = saveFileDialog1.FileName;
            File.WriteAllText(pfad, textBox1.Text);
            this.Text = pfad;
        }
    }
```

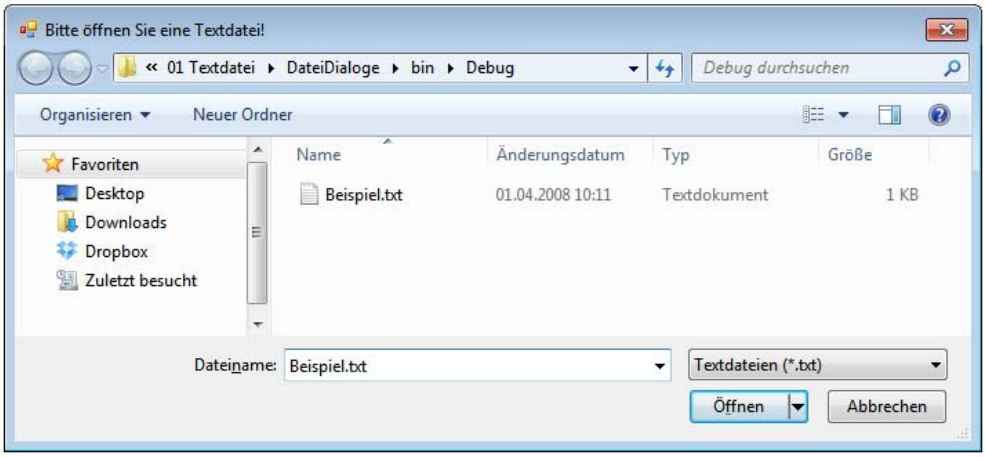
Der Menüpunkt *Datei|Beenden*:

```
    private void beendenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

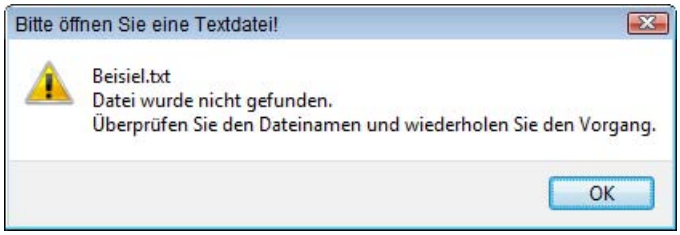
Test

Durch eigene Experimente lässt sich am besten die Wirksamkeit der zahlreichen Eigenschaften der Dateidialoge erkunden:

Beim Öffnen und Speichern wird der Dateipfad in der Titelleiste des Formulars angezeigt:



Warnung bei Eingabe eines nicht vorhandenen Dateinamens im Öffnungsdialog:



8.8.2 Einen Objektbaum persistent speichern

In diesem Beispiel soll gezeigt werden, wie man auch ohne Datenbank und ADO.NET komplexere Datenstrukturen auf der Festplatte speichern kann und wie einfach die Benutzerschnittstelle die Datenbindung mittels *BindingSource*- und *BindingNavigator*-Komponenten realisiert.

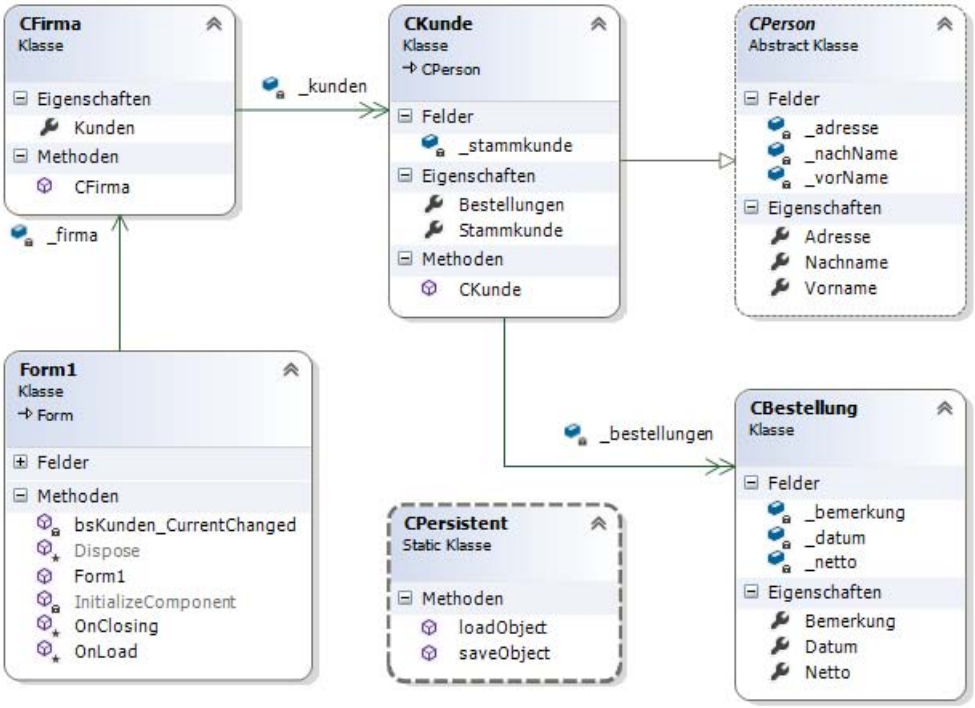
Klassendiagramm

Ausgangspunkt ist das mit dem Klassendesigner von Visual Studio erstellte Klassendiagramm, welches neben dem Startformular *Form1* die Klassen *CFirma*, *CKunde*, *CPerson*, *CBestellung* und *CPersistent* enthält.

Wie Sie sehen, verwaltet die Benutzerschnittstelle *Form1* eine Instanz der Klasse *CFirma,* die über eine multiple Assoziation (Auflistungszuordnung[1]) mit der Klasse *CKunde* verbunden ist (eine Firma kann keinen, einen oder mehrere Kunden haben).

[1] Eine Auflistungszuordnung ist – im Unterschied zur einfachen Zuordnung – im Klassen -Designer an den doppelten Pfeilspitzen erkennbar.

Eine gleichartige Beziehung besteht auch zwischen den Klassen *CKunde* und *CBestellung* (ein Kunde kann keine, eine oder mehrere Bestellungen haben). Man erkennt aus diesen Zusammenhängen, dass sich – ausgehend von einer Instanz von *CFirma* – die Objekte über Auflistungen von *CKunde* zu *CBestellung* baumartig verzweigen.



Die Klasse *CKunde* erbt von der abstrakten Klasse *CPerson*. Weiterhin gibt es eine statische Klasse *CPersistent*, welche Methoden zum Speichern und Laden des Objektbaums – dieser wird von der in *Form1* erzeugten Instanzenvariablen *_firma* gekapselt – bereitstellt.

Wer bereits über Erfahrungen mit dem in Visual Studio integrierten Klassen Designer verfügt (er ist kinderleicht zu bedienen!) kann ihn schon zu Beginn in die Codeentwicklung einbinden und sich dadurch mancherlei Arbeit ersparen, denn der Designer generiert z.B. den Rahmencode für Klassen und Methoden und kapselt Felder zu Eigenschaften. Der Quellcode wird automatisch mit dem Klassendiagramm synchronisiert.

Klasse CBestellung

Jede am Objektbaum beteiligte Klasse muss mit dem *[Serializable]*-Attribut markiert sein.

```
[Serializable]
public class CBestellung
{
     private DateTime _datum;
     private decimal _netto;
```

```
    private string _bemerkung;
    public DateTime Datum
    {   get { return _datum; } set { _datum = value; }  }
    public decimal Netto
    {   get { return _netto; }  set { _netto = value; }  }

    public string Bemerkung
    {   get { return _bemerkung; }  set { _bemerkung = value; }
    }
}
```

Klasse CPerson

Diese abstrakte Klasse stellt ihren Nachkommen vier allgemeine Eigenschaften als "Erbmaterial" zur Verfügung:

```
[Serializable]
public abstract class CPerson
{
    private string _vorName, _nachName, _adresse;

    public string Vorname
    { get { return _vorName; } set { _vorName = value; } }

    public string Nachname
    { get { return _nachName; } set { _nachName = value; } }

    public string Adresse
    { get { return _adresse; } set { _adresse = value; } }
}
```

Klasse CKunde

Die Klasse *CKunde* erbt von *CPerson*. Die Bestellungen des Kunden werden in einer generischen Liste vom Typ *CBestellung* gekapselt.

```
[Serializable]
public class CKunde : CPerson
{
    private bool _stammkunde;
    private IList<CBestellung> _bestellungen;                    // generische Liste
    public CKunde()
    {
        _bestellungen = new List<CBestellung>();
    }

    public bool Stammkunde
    {  get { return _stammkunde; }
       set { _stammkunde = value; }  }
```

```
    public IList<CBestellung> Bestellungen
    {   get { return _bestellungen; }
        set { _bestellungen = value; } }
}
```

Klasse CFirma

Diese Klasse ist die Wurzelklasse des Objektbaums und kapselt lediglich die Kundenliste (generische Liste vom Typ *CKunde*).

```
[Serializable]
public class CFirma
{
    private IList<CKunde> _kunden;                  // generische Liste

    public CFirma()
    {
        _kunden = new List<CKunde>();
    }

    public IList<CKunde> Kunden
    {
        get { return _kunden; }
        set { _kunden = value; }
    }
}
```

Klasse CPersistent

Diese statische Klasse exportiert die Methoden *saveObject* und *loadObject*, mit denen die Serialisierung/-Deserialisierung beliebiger Objekte möglich ist.

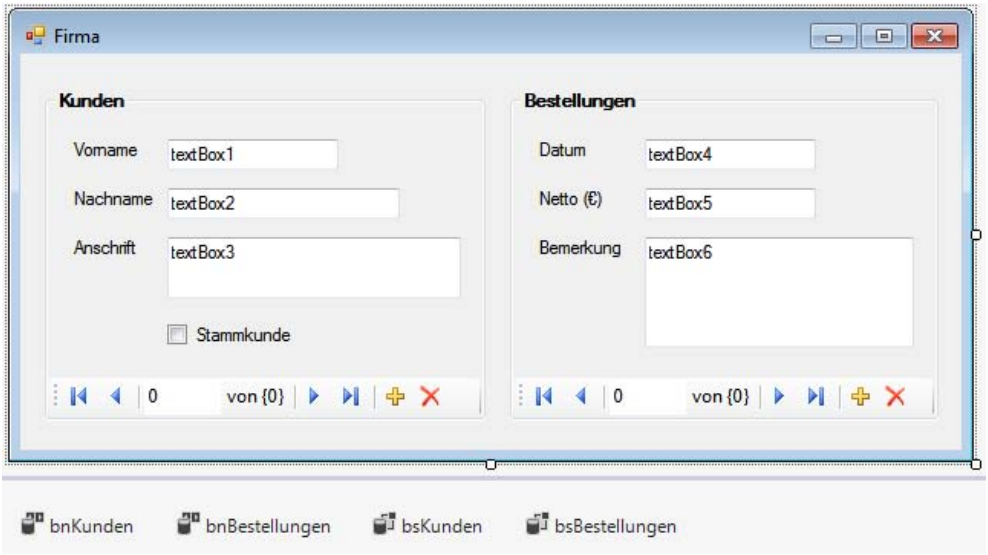
Die Methode *saveObject* übernimmt als Parameter das Objekt und den Dateipfad, serialisiert das Objekt und speichert es auf der Festplatte ab.

Die Methode *loadObject* erwartet als Parameter den Dateipfad, holt sich das Objekt von der Festplatte und liefert es deserialisiert zurück.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public static class CPersistent
{
    public static void saveObject(Object o, string pfad)
    {
        FileStream fs = new FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None);
        BinaryFormatter bf = new BinaryFormatter();
        bf.Serialize(fs, o);
        fs.Close();
    }
```

```
    public static Object loadObject(string pfad)
    {
        FileStream fs = new FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read);
        BinaryFormatter bf = new BinaryFormatter();
        Object o = bf.Deserialize(fs);
        fs.Close();
        return o;
    }
}
```

Form1



Die Bedienelemente zu Kunden und Bestellungen sind in jeweils einer *GroupBox* angeordnet, an deren Fuß eine *BindingNavigator*-Komponente angedockt hat. Weiterhin werden zwei *BindingSource*-Komponenten benötigt.

Verknüpfen Sie im Eigenschaftenfenster die *BindingSource*-Property von *bnKunden* mit *bsKunden* und von *bnBestellungen* mit *bsBestellungen*.

```
public partial class Form1 : Form
{
    ...
```

Man sieht es dieser Variablen nicht an, dass sie den kompletten Objektbaum kapselt:

```
    private CFirma _firma = new CFirma();
```

Der Dateipfad verweist in unserem Fall auf das Ausgabeverzeichnis des Projekts:

```
    private const string PFAD = "Bestellungen.dat";
```

Beim Laden des Formulars wird versucht, die Datei zu laden (falls die Datei nicht vorhanden ist, wird eine neue leere Datei angelegt):

```
protected override void OnLoad(EventArgs e)
{
   try
   {
       _firma = (CFirma) CPersistent.loadObject(PFAD);
   }
   catch (Exception ex)
   {
       MessageBox.Show(ex.Message);
   }
   bsKunden.DataSource = _firma.Kunden;

   textBox1.DataBindings.Add(new Binding("Text", bsKunden, "Vorname", true));
   textBox2.DataBindings.Add(new Binding("Text", bsKunden, "Nachname", true));
   textBox3.DataBindings.Add(new Binding("Text", bsKunden, "Adresse", true));
   checkBox1.DataBindings.Add(new Binding("Checked", bsKunden, "Stammkunde", true));
   base.OnLoad(e);
}
```

Wenn zu einem anderen Kunden gewechselt wird, müssen auch die zu diesem Kunden gehörenden Bestellungen ermittelt und der *BindingSource* zugewiesen werden. Anschließend werden die *TextBox*en erneut angebunden:

```
private void bsKunden_CurrentChanged(object sender, EventArgs e)
{
   CKunde kunde = (CKunde) bsKunden.Current;
   bsBestellungen.DataSource = kunde.Bestellungen;
   textBox4.DataBindings.Clear();
   textBox4.DataBindings.Add("Text", bsBestellungen, "Datum", true);
   textBox5.DataBindings.Clear();
   textBox5.DataBindings.Add("Text", bsBestellungen, "Netto", true);
   textBox6.DataBindings.Clear();
   textBox6.DataBindings.Add("Text", bsBestellungen, "Bemerkung", true);
}
```

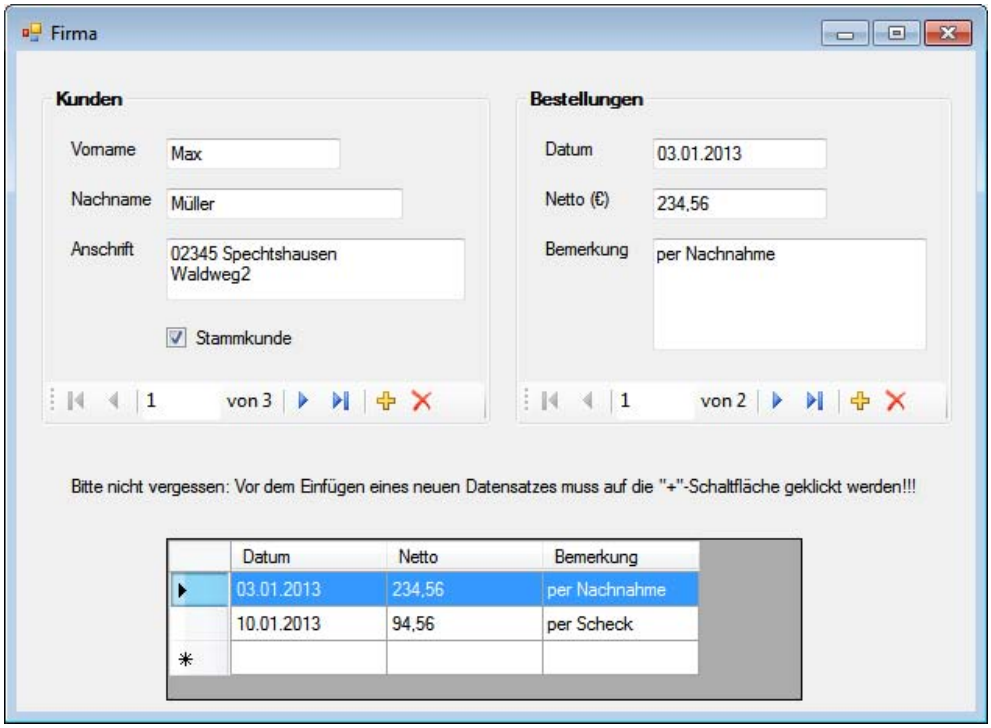
Beim Schließen des Formulars wird der komplette Objektbaum abgespeichert:

```
protected override void OnClosing(CancelEventArgs e)
{
    try
    {
        CPersistent.saveObject(_firma, PFAD);
    }
    catch (Exception ex)
    {   MessageBox.Show(ex.Message); }
    base.OnClosing(e);
}
}
```

Test

Um einen Kunden hinzuzufügen, muss zunächst die "+"-Schaltfläche des *BindingNavigator*s geklickt werden. Analog ist beim Hinzufügen einer Bestellung zu verfahren.

Wundern Sie sich nicht, dass in der folgenden Laufzeitabbildung auch eine *DataGridView*-Komponente enthalten ist (siehe Bemerkung am Schluss).



Die eingegebenen Datensätze gehen nicht verloren, da sie beim Schließen des Formulars automatisch gespeichert werden, um nach einem erneuten Programmstart wieder zur Verfügung zu stehen.

Anzeige mit DataGridView

- Eine einzige Codezeile genügt, um alle Bestellungen eines Kunden in einer zusätzlich hinzugefügten *DataGridView*-Komponente anzuzeigen:

```
dataGridView1.DataSource = bsBestellungen;
```

- Da das *DataGridView* auch editierbar ist, könnte auf die Detailanzeige in den *TextBox*en und auf den zweiten *BindingNavigator* auch verzichtet werden. Gleiches gilt natürlich auch für die Anzeige der Kundenliste.

8.8.3 Ein Memory Mapped File (MMF) verwenden

Ein MMF (siehe Abschnitt 8.7) mappt den Inhalt einer Datei an den logischen Adressraum einer Anwendung und versetzt den Programmierer in die Lage, mit sehr großen Dateien zu arbeiten, weil der Speicher konkurrierend verwaltet werden kann. Außerdem ist ein wahlfreier Zugriff (Random Access) ohne langwierige Suchoperationen möglich.

Das vorliegende Beispiel demonstriert den Einsatz eines MMF beim Manipulieren einer Bilddatei. Ganz nebenbei lernt mancher auch eine völlig andere Bedeutung der *using*-Klausel näher kennen.

Oberfläche

Auf das Startformular einer Windows Forms-Anwendung setzen Sie einen *Button* und eine *PictureBox*.

Quellcode

```
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

public partial class Form1 : Form
{
    ...
```

Eine RGB-Farbstruktur definieren (inkl. einer Methode):

```
    public struct MyRGB
    {
        public short Red;
        public short Green;
        public short Blue;
        public short Alpha;
```

Farbanteile verfälschen (der Einsatz der mathematischen *Min*-Funktion verhindert hier eine Bereichsüberschreitung):

```
        public void changeRGB(short value)
        {
            Red = (short) Math.Min(short.MaxValue, (int) Red + value);
            Green = (short) Math.Min(short.MaxValue, (int) Green + value);
            Blue = (short) Math.Min(short.MaxValue, (int) Blue + value);
            Alpha = (short) Math.Min(short.MaxValue, (int) Alpha + value);
        }
    }
```

Die Start-Schaltfläche:

```
    private void button1_Click(object sender, EventArgs e)
    {
```

Die MMF-Datei für die sich im Anwendungspfad befindliche Datei *Bild1.bmp* wird nun erzeugt. Übergeben werden auch der Zugriffsmodus und der Namen des MMF (im weiteren Verlauf bedeutungslos). Die *using*-Klausel definiert hier einen Bereich, an dessen Ende ein Objekt sofort wieder freigegeben wird.

```
        using (var mmf = MemoryMappedFile.CreateFromFile("Bild1.bmp", FileMode.Open, "MMF1"))
        {
```

Für den Zugriff auf die MMF müssen wir einen Map View definieren, dieser soll den Speicherbereich zwischen den Adressen 1MByte und 11MByte (Offset plus Länge) abdecken.

Die Hex-Adressen werden durch den Präfix *0x* gekennzeichnet:

```
            long offset = 0x100000;     //  1 MB
            long länge = 0xA00000;      // 10 MB

            using (var vacc = mmf.CreateViewAccessor(offset, länge))
            {
```

Anzahl der Bytes pro Farbwert ermitteln:

```
                int colorSize = Marshal.SizeOf(typeof(MyRGB));
```

Eine Puffervariable für den einzulesenden Farbwert:

```
                MyRGB color;
```

Den Map View manipulieren:

```
                for (long i = 0; i < länge; i += colorSize)   //
                {
```

Vom MMF in *color*-Variable einlesen:

```
                    vacc.Read(i, out color);
```

Die *color*-Variable manipulieren:

```
                    color.changeRGB(20);
```

Die *color*-Variable zurück ins MMF schreiben:

```
                    vacc.Write(i, ref color);
                }
            }
        }
```

Bild anzeigen:

```
        pictureBox1.Load("Bild1.bmp");
    }
}
```

Test

Starten Sie das Programm und klicken Sie dann auf die Schaltfläche. Beobachten Sie, wie sich die Farbwerte des Bildes verändert haben.

Bemerkungen

- Für ein MMF können auch mehrere Map Views definiert werden, womit ein einfacher Datenaustausch zwischen verschiedenen Prozessen möglich wird.
- Die *using*-Anweisung ist besonders dann zweckmäßig, wenn es um das sofortige Freigeben von Objekten geht, die viel Speicherplatz belegen (ansonsten erledigt das irgendwann der Garbage Collector).
- Einen praktischen Rechner zur Dezimal-Hexadezimal-MByte-Konvertierung finden Sie in unserem [Visual C# 2012-Kochbuch].

Kapitel 9

Asynchrone Programmierung

In diesem und dem folgenden Kapitel werden wir uns den (positiven und negativen) Aspekten der asynchronen Programmierung mit dem .NET-Framework zuwenden. Wir wollen uns mit

- der Verwendung von Threads,
- den Möglichkeiten von Sperren (*lock*, *Monitor, Mutex)*,
- der *BackgroundWorker*-Komponente,
- dem asynchronen Programmierentwurfsmuster ,
- dem asynchronen Aufruf mittels Delegate,
- dem Einsatz der *Task Parallel Library* (siehe Kapitel 10) und, last but not least,
- den mit Visual Studio 2012 eingeführten neuen Schlüsselwörter *async* und *await*

beschäftigen. Ein komplexes Anwendungsbeispiel rundet das Kapitel ab.

HINWEIS: Auch wenn mit der *Task Parallel Library* viele Aufgaben mit weniger Aufwand gelöst werden können, wollen wir uns zunächst mit den klassischen Varianten der asynchronen Programmierung auseinandersetzen, da die o.g. Library final und komplett erst ab dem .NET-Framework 4, verfügbar ist. Viele bereits bekannte Konzepte, wie z.B. der Zugriff auf den Vordergrund-Thread (Oberfläche), haben sich auch bei Verwendung der *Task Parallel Library* nicht geändert, ein Blick auf diese Details schadet deshalb nicht, sondern verhilft zum dringend erforderlichen Grundverständnis der neueren Technologien.

9.1 Übersicht

Threads stehen im Mittelpunkt der so genannten "asynchronen Programmierung". Darunter wollen wir all jene Technologien verstehen, die es einer Anwendung ermöglichen, Code auszuführen ohne das aktuelle Programm, d h. meist den Vordergrundthread mit der Oberfläche, an dieser Stelle anzuhalten. In der Regel ist dies im Zusammenhang mit dem Warten auf relativ unbestimmte

Ereignisse oder lang andauernde Vorgänge/Berechnungen von Belang. Die Anwendung soll weiter durch den Nutzer bedienbar bleiben, gleichzeitig wird im Hintergrund zum Beispiel eine Grafik konvertiert oder eine Dateioperation abgeschlossen.

Im Zusammenhang mit dem gerade beschriebenen, quasi parallelen, Abarbeiten von Aufgaben durch den Computer tauchen immer wieder einige Begriffe auf, die Sie unbedingt unterscheiden und kennen sollten:

- *Multitasking*
- *Multithreading*
- *Parallel-Programmierung*
- *Deadlocks*
- *Racing*

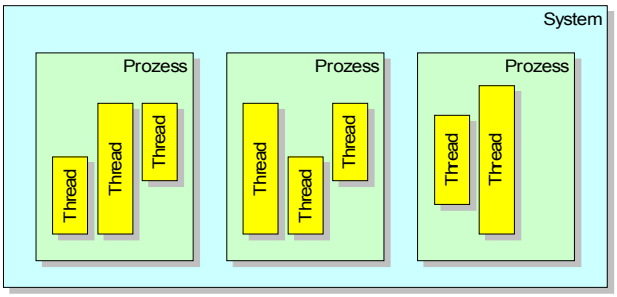
Im Folgenden wollen wir diese Begriffe im Zusammenhang erklären.

9.1.1 Multitasking versus Multithreading

Während beim Multitasking jedem Prozess (eigener Speicherbereich, eigener Heap, eigene Variablen) durch das System eine bestimmte Prozessorzeit zugeteilt wird[1], werden beim Multithreading einzelne Ablaufstränge (Fäden → Threads) **innerhalb** des Prozesses gesteuert.

HINWEIS: Leider taucht im Zusammenhang mit der neueren *Task Parallel Library* auch eine andere Bedeutung des Begriffs "Task" auf. Hier ist die einzelne **Aufgabe** innerhalb eines Programms (Prozesses) gemeint, derartige *Task*-Objekte sind eigentlich Threads, die um ein paar Verwaltungsfunktionen (Scheduler) bereichert wurden. Es handelt sich definitiv nicht um die vom System-"Taskmanager" verwalteten Prozesse. Wir gehen auf diese Thematik allerdings erst im Kapitel 10 ein.

Die folgende Abbildung zeigt die grundsätzliche Aufteilung (unterschiedlich viele Threads können zu unterschiedlichen Zeiten innerhalb der Prozesse ausgeführt werden).



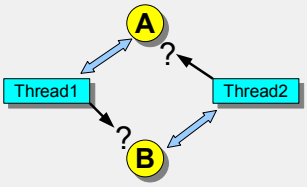
[1] Echte parallele Ausführung ist nur auf Mehrprozessorsystemen möglich.

9.1.2 Deadlocks

Mit diesem Begriff wird ein Szenario bezeichnet, in welchem mindestens zwei Threads gegenseitig aufeinander oder auf die Freigabe eines Objekts warten, ohne je zu einem Ende zu kommen.

BEISPIEL 9.1: Deadlock

Thread1 möchte die Objekte *A* und *B* bearbeiten und zu diesem Zweck sperren. *Thread2* hat die gleiche Aufgabe, sperrt jedoch erst Objekt *B* und dann Objekt *A*. Kommt es durch die Verteilung der Prozessorzeit dazu, dass *Thread1* Objekt *A* gesperrt hat und nachfolgend *Thread2* Objekt *B* sperrt, wartet *Thread1* bis in alle Ewigkeit auf die Freigabe von Objekt *B* und *Thread2* wartet ebenfalls auf die Freigabe von Objekt *A*.



Lösen lässt sich dieses Problem zum Beispiel durch eine konsequente Festlegung der Reihenfolge für die einzelnen Sperren oder eine maximale Wartezeit für das Sperren von Objekten.

9.1.3 Racing

Racing bezeichnet den Vorgang, wenn zwei Threads auf die gleiche Ressource (z.B. Datei, Ausgabegerät, Variable etc.) zugreifen wollen. Durch unterschiedlich schnelle Verarbeitung der Threads kommt es zu undefinierten Zuständen, d h. es ist nicht vorhersehbar, welcher Thread das "Rennen" gewinnt.

Derartige Probleme können lange unentdeckt bleiben, bis zum Beispiel die Anwendung auf ein Mehrprozessorsystem verlagert wird oder das System sehr stark ausgelastet ist, was zu einer verlangsamten Abarbeitung einzelner Threads führt. Erst unter diesen Umständen wird aus dem bisherigen korrekten Verhalten, ein Fehler, der meist schwer reproduzierbar ist.

BEISPIEL 9.2: Racing

C#

Ein kleines Testprogramm (mit der *Task Parallel Library*, siehe ab Seite 503) zeigt die Problematik eines "Wettrennens" am Beispiel der Ausgabekonsole.

```
using System.Threading;
using System.Threading.Tasks;
...
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start");
```

BEISPIEL 9.2: Racing

Zwei Tasks (Threads) erzeugen und starten:

```
            var thread1 = Task.Factory.StartNew(MethodeA);
            var thread2 = Task.Factory.StartNew(MethodeB);
```

Hier synchronisieren wir und warten bis **beide** Threads beendet sind:

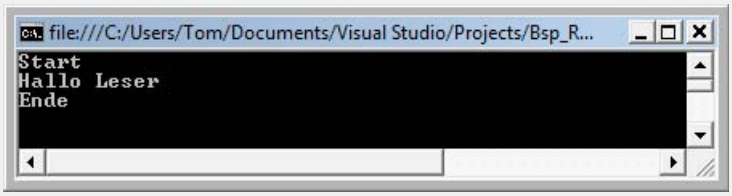
```
            Task.WaitAll(thread1, thread2);
            Console.WriteLine("Ende");
            Console.Read();
        }
```

Die beiden "anspruchsvollen" Methoden für die jeweiligen Threads:

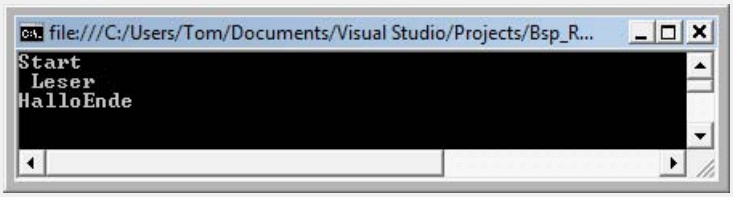
```
        static void MethodeA()
        { Console.Write("Hallo"); }

        static void MethodeB()
        { Console.WriteLine(" Leser"); }
    }
}
```

Ergebnis

Der Normalfall zeigt folgende Ausgabe:



Aber nach x Programmstarts finden Sie statt obigem Ergebnis die folgende Ausgabe vor:



In diesem Fall war *Thread2*, also *MethodeB*, etwas schneller, obwohl *Thread1* vorher erzeugt und gestartet wurde. Beachten Sie jedoch, dass Ihr Programm nicht unabhängig vom restlichen System agiert, auch andere Prozesse haben Einfluss auf die Verteilung der Prozessorzeit und so kann es sein, dass *Thread1* erst einmal in eine kleine Warteschleife geschickt wurde und *Thread2* so günstig erzeugt wurde, dass er sofort ausgeführt werden konnte.

In obigem Beispiel handelt es sich lediglich um ein triviales Problem, bei Datenbankabfragen bzw. Einfüge-Operationen kann ein Racing schnell zum Show-Stopper werden und Ihre Geduld bei der Fehlersuche auf eine harte Probe stellen.

HINWEIS: Die Lösung im obigen Fall wäre eine Synchronisierung von *Thread2* auf *Thread1,* allerdings ist dann das Beispiel sinnfrei, da wir *MethodeB* auch gleich in *MethodeA* integrieren könnten.

9.2 Programmieren mit Threads

Die Multithreading-Funktionalität, d.h. die dafür notwendigen Klassen, werden vom Namespace *System.Threading* bereitgestellt. Im Mittelpunkt steht die Klasse *Thread*, mit der Sie Threads erzeugen und steuern können.

9.2.1 Einführungsbeispiel

Bevor wir Sie mit trockenen Ausführungen langweilen, zunächst ein kleines Beispiel der Thread-Programmierung.

BEISPIEL 9.3: Verwendung der *Thread*-Klasse um eine Methode Berechne als eigenständigen Thread auszuführen. Zusätzlich übergeben wir noch einen Parameter (*anzahl*) an die Methode.

C#

```
...
using System.Threading;
using System.Diagnostics;
using System.Media;

namespace Thread1
{
    public partial class Form1 : Form
    {
        private void button1_Click(object sender, EventArgs e)
        {
```

Einen neuen Thread erzeugen (wir übergeben einen Delegate, d h. die Methode, die als eigener Thread ausgeführt werden soll):

```
            Thread myThread = new Thread(Berechne);
            Debug.Print("Thread Gestartet");
```

Wir starten den Thread und übergeben bei dieser Gelegenheit gleich noch einen Parameter:

```
            myThread.Start(15);
            Debug.Print("Nach Thread.Start");
        }
...
```

BEISPIEL 9.3: Verwendung der *Thread*-Klasse um eine Methode Berechne als eigenständigen Thread auszuführen. Zusätzlich übergeben wir noch einen Parameter (*anzahl*) an die Methode.

Nach dem Aufruf der Methode *Start* wird die Programmausführung asynchron fortgesetzt, d h., noch vor dem Ende des Threads wird die nachfolgende Anweisung ausgeführt.

Die eigentliche Arbeit verrichtet die folgenden Methode:

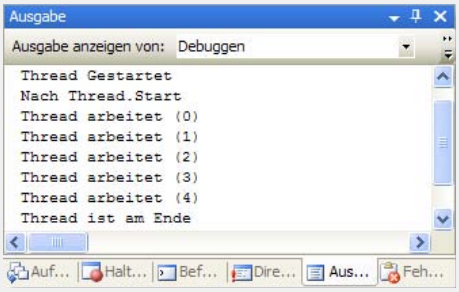
```
private void Berechne(object anzahl)
{
```

Eine Schleife, um den Thread etwas warten zu lassen:

```
    for (int i = 0; i < (int) anzahl; i++)
    {
        Thread.Sleep(1000);
        Debug.Print("Thread arbeitet ({0})", i);
        SystemSounds.Beep.Play();
    }
    Debug.Print("Thread ist am Ende");
}
```

Ergebnis

Nach dem Start wird im Ausgabefenster der Text "Thread Gestartet" erscheinen, sofort darauf "Nach Thread.Start" (es ist nun mal eine asynchrone Ausführung) und danach die Meldungen aus dem eigentlichen Thread:

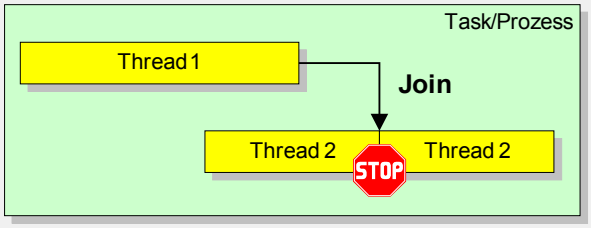


Ganz nebenbei werden Sie aus dem Lautsprecher mit Tönen gefoltert, aber so schlafen Sie beim Ausprobieren dieses Trivialbeispiels wenigstens nicht ein.

9.2.2 Wichtige Thread-Methoden

Mit einigen Methoden der *Thread*-Klasse hatten wir es im vorhergehenden Beispiel bereits zu tun, nun zu den Einzelheiten:

Methode	Beschreibung
Thread	Der Konstruktor erwartet einen Delegate, der auf die auszuführende Methode verweist. Dieser Delegate kann parameterlos sein oder die Übergabe eines *Object*-Typs ermöglichen.

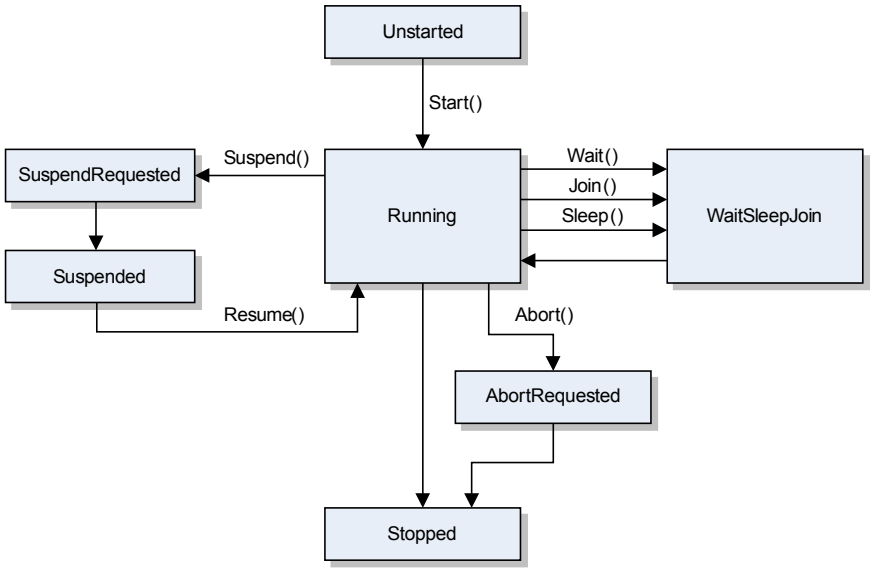
Methode	Beschreibung
Start	Die Ausführung des Threads beginnt. Vorher haben Sie noch die Möglichkeit, zum Beispiel den Namen oder die Priorität des Threads festzulegen.
Suspend	Hält den Thread an. Der Thread verbraucht keine Prozessorzeit mehr. Achtung: Diese Methode ist veraltet!
Resume	Setzt die Ausführung des Threads fort, egal wie oft vorher *Suspend* aufgerufen wurde. Achtung: Diese Methode ist veraltet!
Interrupt	Unterbricht einen Thread, der sich im Wait-, Sleep- oder Join-Zustand befindet. Hierdurch wird eine *ThreadInterruptedException* ausgelöst, die Sie abfangen sollten.
Join	Blockiert die Ausführung des aktuellen Threads, bis der angegebene Thread beendet ist. 
Sleep	Hält die Ausführung des Threads für eine vorgegebene Anzahl von Millisekunden oder eine Zeitspanne (*TimeSpan*) an, bevor die Ausführung automatisch fortgesetzt wird. Der Thread verbraucht währenddessen keine Prozessorzeit. Achtung: Sie können nur *Thread.Sleep* aufrufen. Dies bezieht sich immer auf den aktuellen Thread.
Abort	Beginnt den Abbruch des Threads. Ein erneuter Start ist nicht mehr möglich.

HINWEIS: Generell kann ein Thread nach seiner Abarbeitung nicht wieder gestartet werden. Sie müssen eine neue Instanz erzeugen und diese starten.

HINWEIS: Wenn Sie auf Codebeispiele treffen, bei denen im Konstruktoraufruf noch ein *ThreadStart*- oder *ParameterizedThreadStart*-Delegate erzeugt wird, vergessen Sie diese einfach, denn das übernimmt der C#-Compiler für Sie.

```
private void button1_Click(object sender, EventArgs e)
{
    this.myThread = new Thread(new ParameterizedThreadStart(this, (IntPtr) this.Berechne));
    this.myThread.Name = "Test-Thread";
    this.myThread.IsBackground = true;
    this.myThread.Start(15);
}
```

Eine Übersicht, wie Sie die Methoden einsetzen um zwischen den einzelnen Zuständen des Threads zu wechseln, zeigt die folgende Abbildung:



HINWEIS: Mehrere Zustände können gleichzeitig aktiv sein (z.B. *WaitSleepJoin* und *AbortRequested*).

9.2.3 Wichtige Thread-Eigenschaften

Folgende Eigenschaften sind von Interesse:

Eigenschaft	Beschreibung
IsAlive	*true*: Der Thread ist gestartet und noch nicht beendet. Dieser Wert ist auch *true*, wenn sich der Thread im Sleep-Zustand befindet.
Name	Ein Name für den Thread, diesen können Sie aus dem Thread heraus mit *Thread.CurrentThread.Name* auslesen. Diese Eigenschaft ist beim Debuggen ganz nützlich, um die einzelnen Threads voneinander zu unterscheiden.
Priority	Die Ausführungspriorität des Threads: *Highest*, *AboveNormal*, *Normal*, *BelowNormal*, *Lowest.*
ThreadState	Der aktuelle Status des Threads: *Aborted*, *AbortRequested*, *Background*, *Running*, *Stopped*, *StopRequested*, *Suspended*, *SuspendRequested*, *Unstarted*, *WaitSleepJoin.*

Eigenschaft	Beschreibung
IsBackGround	Es handelt sich um einen Hintergrund-Thread, d.h., der Thread wird beim Beenden der Anwendung (Vordergrund-Thread) automatisch von der CLR beendet. Sie können diese Eigenschaft für Ihre Threads selbst festlegen. Ist die Eigenschaft auf *false* gesetzt, wird die Anwendung erst nach dem Ende des Threads beendet. Dies kann zu unerwünschten Nebeneffekten führen, wenn Sie zu diesem Zeitpunkt noch auf den Vordergrund-Thread zugreifen wollen.

HINWEIS: Eine umfangreiche Anwendung, in welcher Sie den Einsatz der *Thread*-Klasse "spielend" lernen, finden Sie im Praxisbeispiel 9.10.1!

Stehen Sie vor der Aufgabe, viele Anforderungen gleichzeitig zu bearbeiten und nehmen diese nicht allzu viel Zeit in Anspruch, können Sie entweder selbst eine Collection von Threads erstellen, oder Sie nutzen gleich die Vorteile eines vorhandenen Threadpools.

Bleibt es dem Anwender oder der Applikation überlassen, wie viele Threads angefordert werden können, kann es schnell zu einem Performance-Einbruch kommen, denn die meiste Prozessorzeit wird für die Verwaltung der Threads verbraucht und nicht für deren Bearbeitung. Anders beim Threadpool, hier ist die Anzahl der Threads begrenzt, neue Threads werden erst erstellt, wenn ein Thread wieder frei ist.

HINWEIS: Auf das explizite Erzeugen und Parametrieren der Threads müssen Sie hier zugunsten ihrer einfacheren Verwendung allerdings verzichten.

9.2.4 Einsatz der ThreadPool-Klasse

BEISPIEL 9.4: Anwendung eines ThreadPools

C#

Die eigentliche *Thread*-Methode:

```
public void Berechne(object obj)
{
   int n1, n2, n3;
   Debug.Print("-> Thread {0} gestartet", (int) obj);
   for (int i = 0; i < 10; i++)
   {   Thread.Sleep(100); }
```

Abrufen der maximalen Anzahl von Threads:

```
   ThreadPool.GetMaxThreads(out n3, out n2);
```

Abrufen der noch verfügbaren Threads:

```
   ThreadPool.GetAvailableThreads(out n1, out n2);
   Debug.Print("  <- Thread {0} beendet! (frei {1} von {2})", (int) obj, n1 + 1, n3);
}
```

BEISPIEL 9.4: Anwendung eines ThreadPools

Wir addieren 1, da der aktuelle Thread ja auch gleich beendet wird.

Das eigentliche Verwenden des Threadpools:

```
private void button1_Click(object sender, EventArgs e)
{
   int n2, n3;
```

Maximale Anzahl von Threads bestimmen:

```
   ThreadPool.GetMaxThreads(out n3, out n2);
```

Wir erzeugen ein paar mehr Threads als in den Threadpool passen:

```
   for (int i = 0; i < (n3+10); i++)
   {
      ThreadPool.QueueUserWorkItem(Berechne,i);
      Debug.Print("Thread_{0} in Warteschlange aufgenommen",i);
   }
}
```

Die Ausführung beginnt mit dem Aufruf der Methode *QueueUserWorkItem*. Rufen Sie die Methode wie oben auf, werden lediglich die Threads aus dem Pool verwendet. Die restlichen Anforderungen werden erst dann bearbeitet, wenn wieder ein Thread aus dem Threadpool verfügbar ist.

Ergebnis

Starten Sie das Programm (siehe Beispieldaten), werden zunächst alle Anforderungen in die Warteschlange gestellt, die ersten Threads werden bearbeitet.

Auszug aus dem Protokoll des Ausgabefensters:

```
Thread 0 in Warteschlange aufgenommen
Thread 1 in Warteschlange aufgenommen
Thread 2 in Warteschlange aufgenommen
Thread 3 in Warteschlange aufgenommen
-> Thread 0 gestartet
Thread 4 in Warteschlange aufgenommen
...
Thread 59 in Warteschlange aufgenommen
-> Thread 2 gestartet
  <- Thread 0 beendet! (frei 48 von 50)
-> Thread 3 gestartet
  <- Thread 1 beendet! (frei 48 von 50)
-> Thread 4 gestartet
-> Thread 5 gestartet
-> Thread 6 gestartet
  <- Thread 2 beendet! (frei 46 von 50)
-> Thread 7 gestartet
...
  <- Thread 55 beendet! (frei 46 von 50)
```

BEISPIEL 9.4: Anwendung eines ThreadPools

```
<- Thread 56 beendet! (frei 47 von 50)
<- Thread 57 beendet! (frei 48 von 50)
<- Thread 58 beendet! (frei 49 von 50)
<- Thread 59 beendet! (frei 50 von 50)
```

Nachdem wieder Threads frei sind, werden die restlichen Anforderungen bearbeitet.

9.3 Sperrmechanismen

Nicht in jedem Fall sind Threads ein Segen für den Programmierer. Da einzelne Methodenaufrufe oder Operationen wiederum auf unzähligen Maschinenanweisungen basieren können und die Threadumschaltung zu unvorhergesehenen Zeitpunkten erfolgen kann, sind unter gewissen Umständen undefinierte Zustände möglich und das Programm verhält sich nicht wie gewünscht. Meist ist dieses Verhalten nicht reproduzierbar, da weitere Prozesse und Systemeinstellungen Einfluss auf die Rechenzeitverteilung haben.

In diesem Fall ist es sinnvoll, die Threadumschaltung so lange zu unterbrechen, bis die kritischen Operationen ausgeführt wurden. Das .NET-Framework stellt dafür verschiedene Mechanismen bereit.

9.3.1 Threading ohne lock

BEISPIEL 9.5: Wir erzeugen zwei Threads, die auf die gleiche Variable *summe* zugreifen. Ein Thread versucht zu inkrementieren, der andere zu dekrementieren.

C#

```
using System.Threading;
using System.Diagnostics;

namespace ThreadLocking
{
    public partial class Form1 : Form
    {
        public int summe = 0;
```

Die Thread-Methode:

```
        public void Berechne(object anzahl)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread.Sleep(100);
```

Auch wenn die folgende Anweisung im C#-Code trivial aussieht, auf Maschinenebene wird diese Anweisung in mehreren Schritten ausgeführt (Register laden, Verarbeitung, Register auslesen), was dazu führt, dass diese Anweisung, bedingt durch das Multithreading, auch

BEISPIEL 9.5: Wir erzeugen zwei Threads, die auf die gleiche Variable *summe* zugreifen. Ein Thread versucht zu inkrementieren, der andere zu dekrementieren.

mitten in der Operation unterbrochen werden kann (man spricht von einer **nicht** atomaren Operation). In diesem Fall sind Rechenfehler bei konkurrierendem Zugriff auf dieselbe Variable *(summe)* vorprogrammiert.

```
                summe += (int) anzahl;
                Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe);
            }
        }
```

Erzeugen zweier Threads:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Thread t1 = new Thread(Berechne);
            t1.Name = "T1";
            Thread t2 = new Thread(Berechne);
            t2.Name = "T2";
            t1.Start(1);
            t2.Start(-1);
        }
...
```

Ergebnis

Das Ergebnis im Ausgabefenster:

```
Thread T2: Summe = 1
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Der Thread 0x5a4 hat mit Code 0 (0x0) geendet.
Thread T1: Summe = 1
Der Thread 0xec8 hat mit Code 0 (0x0) geendet.
```

Beim erneuten Aufruf ist der Wert von *summe* plötzlich *0*, was eigentlich auch korrekt ist (das klassische Problem, wenn zwei das Gleiche machen wollen). Die Ursache ist, wie schon beschrieben, dass die Threadumschaltung zu so ungünstigen Zeitpunkten erfolgen kann, dass zwar die Prozessorregister mit dem Variableninhalt geladen wurden, aber noch nicht das Inkrementieren oder Dekrementieren erfolgt ist.

HINWEIS: Sollten Sie dieses Problem auch bei mehrfachem Start nicht reproduzieren können, erhöhen Sie einfach mal die Prozessorlast, indem Sie zum Beispiel ein oder zwei HD-Videos im Hintergrund laufen oder rendern lassen.

9.3.2 Threading mit lock

Unter C# bietet sich als einfachste Möglichkeit das *lock*-Schlüsselwort an, das den folgenden Anweisungsblock als kritischen Abschnitt markiert. Das Ergebnis dieser Bemühungen: nur ein Thread kann gleichzeitig auf diesen Abschnitt zugreifen, alle anderen müssen warten, bis das Objekt (dieses geben Sie im *lock*-Statement an) freigegeben wird.

BEISPIEL 9.6: Threading mit *lock*

C#

```
public void BerechneSicher(object anzahl)
{
    lock (this)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(100);
            summe += (int)anzahl;
            Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe);
        }
    }
}
```

Ergebnis

Das Ergebnis im Ausgabefenster:

```
Thread T2: Summe = -1
Thread T2: Summe = -2
Thread T2: Summe = -3
Thread T2: Summe = -4
Thread T2: Summe = -5
Der Thread 0x86c hat mit Code 0 (0x0) geendet.
Thread T1: Summe = -4
Thread T1: Summe = -3
Thread T1: Summe = -2
Thread T1: Summe = -1
Thread T1: Summe = 0
Der Thread 0x414 hat mit Code 0 (0x0) geendet.
```

Zunächst kommt der erste Thread *T2* zum Zug, dann *T1*, aber nicht beide durcheinander und schon garnicht gleichzeitig.

HINWEIS: Ob erst *T1* und dann *T2* ausgeführt wird, oder umgekehrt, ist nicht vorhersagbar, beim erneuten Aufruf kann es schon wieder umgekehrt sein!

Allerdings ist obige Lösung aus zeitlicher Sicht nicht optimal, da zunächst fünf mal *T1* und nachfolgend fünf mal *T2* abgearbeitet wird. Die Pausen (*Thread.Sleep*) summieren sich, was wohl kaum im Sinne des Erfinders ist. Besser ist deshalb folgende Lösung, bei der lediglich das Summieren und die Ausgabe in das *lock*-Statement eingeschlossen werden.

BEISPIEL 9.7: Bessere Lösung

C#

```
...
      for (int i = 0; i < 5; i++)
      {
          Thread.Sleep(100);
          lock (this)
          {
             summe += (int)anzahl;
             Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe);
          }
      }
```

Ergebnis

In diesem Fall können beide Threads fast parallel ausgeführt werden, während der Pause des einen kann der andere arbeiten. Die Ausgabe im Debug-Fenster:

```
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
Thread T2: Summe = 0
Thread T1: Summe = 1
The thread 'T1' (0x126c) has exited with code 0 (0x0).
Thread T2: Summe = 0
The thread 'T2' (0x17d0) has exited with code 0 (0x0).
```

Beim Test werden Sie durch die quasi parallele Abarbeitung eine Beschleunigung der Ausgabegeschwindigkeit bemerken.

HINWEIS: Wie Sie sehen, ist es mit der schnellen Verwendung von gesperrten Abschnitten nicht getan, schnell machen Sie die Vorzüge von Threads wieder zunichte. Beschränken Sie diese Sperren auf die unbedingt notwendige Anzahl und nehmen Sie gegebenenfalls Zeitmessungen vor.

In obigem Beispiel könnten Sie auch die Anzeige aus dem *lock*-Statement herausnehmen, die Berechnung würde in diesem Fall zwar korrekt ablaufen, die Ausgabe auf dem Bildschirm wäre jedoch fehlerhaft.

Ein anschauliches Beispiel

Bereits an dieser Stelle wollen wir Sie an ein Alltagsbeispiel heranführen, das wir in den weiteren Abschnitten noch ausbauen wollen, um auf anschauliche Weise die Funktionsweise der verschiedenen Sperrmechanismen zu verdeutlichen:

Vergleichsbeispiel

Stellen Sie sich zwei oder mehrere Gäste einer Bar als Threads vor, die sich am Tresen einen Drink holen und diesen natürlich auch austrinken wollen. Sie können natürlich alle gleichzeitig auf den Barkeeper (Prozessor) einstürmen, die Folge wird sein, dass kaum einer den gewünschten Drink erhält und die Ausgabe auch recht lange dauern kann (kein Sperrmechanismus).

Alternativ könnten Sie auch dem Barkeeper zurufen, dass Sie einen Drink wünschen. In dieser Zeit ist der Barkeeper für andere Wünsche nicht erreichbar. Allerdings ist es wenig sinnvoll sich so lange mit dem Barkeeper zu unterhalten, bis Sie das nächste Glas bestellen wollen. Besser ist es, wenn Sie auch die anderen Gäste zum Zuge kommen lassen (möglichst kurze Sperre).

9.3.3 Die Monitor-Klasse

Gehen Ihnen die Möglichkeiten des *lock*-Statements nicht weit genug, verwenden Sie doch einen *Monitor*. Auch diese Klasse findet sich im Namenspace *System.Threading*. Auf das Instanziieren können Sie verzichten, da alle Methoden statisch deklariert sind:

- *Enter*
- *TryEnter*
- *Exit*
- *Pulse*
- *PulseAll*
- *Wait*

Die Verwendung scheint zunächst relativ einfach: Mit *Monitor.Enter* markieren Sie den Beginn eines kritischen Abschnitts, d h., Sie sperren diesen Abschnitt für alle anderen Threads. Mit *Monitor.Exit* wird das Ende dieses Abschnitts gekennzeichnet.

HINWEIS: Reduzieren Sie die Länge des Abschnitts auf das unbedingt Notwendige, anderenfalls ist es besser, die Threads mit *Join* zu synchronisieren.

BEISPIEL 9.8: Ausschnitt aus dem Praxisbeispiel 9.10.1 (Spieltrieb & Multithreading erleben)

C#

```
public void Beladen()
{
    Monitor.Enter(this);
    Wartezeit = 0;
    Ladung++;
    myForm.FormRefresh("Schiff: Beladen (" + Ladung.ToString() + ")");
    if (Ladung == 5) Transport();
    Monitor.Exit(this);
}
```

BEISPIEL 9.8: Ausschnitt aus dem Praxisbeispiel 9.10.1 (Spieltrieb & Multithreading erleben)

Tritt innerhalb des markierten Abschnitts ein Laufzeitfehler auf und wird dieser nicht abgefangen, wird auch *Monitor.Exit* nie aufgerufen, und die anderen Threads bekommen keine Möglichkeit, auf den Abschnitt zuzugreifen.

Deshalb besser so:

```
...
     Monitor.Enter(this);
     try
    ...
     finally
       { Monitor.Exit(this); }
...
```

HINWEIS: Auch wenn Sie ein *lock*-Statement verwenden: Intern erzeugt der Compiler Code, der mittels *Monitor*-Klasse und zusätzlicher Fehlerbehandlung diesen Abschnitt absichert. Allerdings können Sie in diesem Fall nicht von den zusätzlichen Möglichkeiten der *Monitor*-Klasse Gebrauch machen.

Wait und Pulse/PulseAll

Neben der direkten Umsetzung des *lock*-Statements über die Methoden *Enter* und *Exit* der *Monitor*-Klasse sind auch die Methoden *Wait* und *Pulse* für den einen oder anderen Spezialfall interessant. Auch hier soll eine "Bar"-Analogie die Funktionsweise verdeutlichen:

Vergleichsbeispiel

Stellen Sie sich vor, Sie sind am Tresen, haben den Drink bestellt und wollen kurzfristig tanzen gehen. In diesem Fall können Sie dem Barkeeper sagen, dass er etwas warten soll (Methode *Wait*). Der Barkeeper kann jetzt die Wünsche anderer Gäste erfüllen. Doch wie kommen Sie an Ihren bereits bestellten Drink? Drei Möglichkeiten bestehen: Sie warten darauf, dass der Gast, der gerade am Tresen steht (und quasi den Monitor besitzt), Sie persönlich darauf hinweist, dass Ihr Drink da ist (Methode *Pulse*). Alternativ kann dieser Gast auch in den Raum rufen, dass noch Gläser abgeholt werden müssen (Methode *PulseAll*). Die dritte Variante: Sie sagen dem Barkeeper vorher Bescheid, dass Sie in einer bestimmten Zeit wieder da sein werden um den Drink abzuholen (Übergabe eines Zeitwertes an die *Wait*-Methode).

Nun aber weg vom Alkohol und hin zur Praxis ...

BEISPIEL 9.9: Verwendung *Wait* und *Pulse*

C#

Zwei Methoden *BerechneSicherMonitorA* und *BerechneSicherMonitorB* sollen konkurrierend ausgeführt werden. Beide greifen auf die Variable *Summe* zu. Zu diesem Zweck wird eine entsprechende Monitor-Sperre verwendet. So weit, so gut. Doch nach dem fünften Schleifendurchlauf in *BerechneSicherMonitorA* soll diese Methode angehalten werden (Methode *Wait*), um zunächst der Methode *BerechneSicherMonitorB* die Vorfahrt zu lassen.

BEISPIEL 9.9: Verwendung *Wait* und *Pulse*

Hat diese 50 Schleifendurchläufe absolviert, heben wir die Sperre für die Methode *BerechneSicherMonitorA* mit *Pulse* wieder auf. Jetzt laufen beide Methoden im Wechsel ab.

Methode *BerechneSicherMonitorA*:

```
public void BerechneSicherMonitorA(object anzahl)
{
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(100);
        Monitor.Enter(this);
        summe += (int)anzahl;
```

Nach fünf Durchläufen warten wir zunächst (hier könnten Sie auch einen Zeitwert übergeben):

```
        if (i == 5) Monitor.Wait(this);
        Debug.Print("Thread {0}: i = {1}  Summe = {2}", Thread.CurrentThread.Name,
                    i, summe);
        Monitor.Exit(this);
    }
}
```

Methode *BerechneSicherMonitorB:*

```
public void BerechneSicherMonitorB(object anzahl)
{
    for (int i = 0; i < 100; i++)
    {
        Thread.Sleep(100);
        Monitor.Enter(this);
        summe += (int)anzahl;
```

Nach 50 Durchläufen wird die Sperre in *BerechneSicherMonitorA* wieder aufgehoben:

```
        if (i == 50) Monitor.Pulse(this);
        Debug.Print("Thread {0}: i = {1}  Summe = {2}", Thread.CurrentThread.Name,
                    i, summe);
        Monitor.Exit(this);
    }
}
```

Ergebnis

Die Ausgabe (in Auszügen):

```
Thread T2: i = 0  Summe = -1
Thread T1: i = 0  Summe = 0
Thread T2: i = 1  Summe = -1
Thread T1: i = 1  Summe = 0
Thread T1: i = 2  Summe = 1
Thread T2: i = 2  Summe = 0
```

BEISPIEL 9.9: Verwendung *Wait* und *Pulse*

```
Thread T1: i = 3  Summe = 1
Thread T2: i = 3  Summe = 0
Thread T1: i = 4  Summe = 1
```

Ab hier ist T1 angehalten, nur noch T2 arbeitet *(Wait)*.

```
Thread T2: i = 4  Summe = 0
Thread T2: i = 5  Summe = 0
Thread T2: i = 6  Summe = -1
...
Thread T2: i = 50  Summe = -45
```

Und hier läuft T1 wieder los *(Pulse)*:

```
Thread T1: i = 5  Summe = -45
Thread T2: i = 51  Summe = -46
Thread T1: i = 6  Summe = -45
Thread T2: i = 52  Summe = -46
...
Thread T2: i = 98  Summe = -89
Thread T2: i = 99  Summe = -90
The thread 'T2' (0x11c0) has exited with code 0 (0x0).
```

Das korrekte Endergebnis ist -90 (10 Schleifendurchläufe mit Addition und 100 Schleifendurchläufe mit Subtraktion). Über *Wait* und *Pulse* haben wir lediglich die Rechenzeit zwischen den Threads anders aufgeteilt.

TryEnter

Sicher ist Ihnen auch schon aufgefallen, dass je nach Anwendungsfall ein *Monitor.Enter* eine recht lange Wartezeit nach sich ziehen kann. Auch hier wollen wir wieder auf unsere "Bar"-Analogie zurückkommen:

Vergleichsbeispiel

TryEnter können Sie mit einem Blick zum Tresen der Bar vergleichen. Ist der Barkeeper nicht beschäftigt, können Sie Ihre Wünsche sofort loswerden. Alternativ bietet *TryEnter* auch die Möglichkeit, einige Zeit zu warten, d h., Sie setzen sich an den Tresen und warten x Minuten, ob Sie bedient werden. Ist dies nicht der Fall, gehen Sie einfach wieder weg.

9.3.4 Mutex

Neben dem *Monitor*-Objekt bietet sich auch ein *Mutex*[1] für das Locking von Codeabschnitten an. Win32-Programmierern wird dieser Begriff sicher bekannt vorkommen, denn ein analoger Mechanismus war auch dort vorhanden.

[1] *mutually exclusive*

HINWEIS: Im Unterschied zum *Monitor* kann ein *Mutex* auch prozessübergreifend verwendet werden.

Im Unterschied zum *Monitor*-Objekt müssen Sie hier jedoch zunächst eine Instanz erzeugen. Dazu stehen Ihnen fünf überladene Konstruktoren zur Verfügung:

```
Mutex ()
Mutex (Boolean)
Mutex (Boolean, String)
Mutex (Boolean, String, Boolean)
Mutex (Boolean, String, Boolean, MutexSecurity)
```

Hier können Sie bereits bestimmen, ob der aktuelle Thread gleich zum Besitzer des Mutex wird und welcher Name verwendet wird.

Leiten Sie den entsprechenden Abschnitt mit der *WaitOne*-Methode ein, um den Mutex zu erhalten. Freigeben können Sie den Mutex mit der *ReleaseMutex*-Methode.

BEISPIEL 9.10: Anwendung eines Mutex

C#

Mutex deklarieren:

```
    public partial class Form1 : Form
    {
        public Mutex myMutex = new Mutex();
...
```

Mutex verwenden:

```
      public void BerechneMutex(object anzahl)
       {
           myMutex.WaitOne();
           for (int i = 0; i < 10; i++)
           {
               Thread.Sleep(200);
               Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe);
           }
           myMutex.ReleaseMutex();
       }
```

Aufruf der Threads (es kann mehrfach auf den Button geklickt werden):

```
private void button4_Click(object sender, EventArgs e)
{

    Thread t1 = new Thread(BerechneMutex);
           t1.Name = "T" + Id.ToString();
           Id++;
           t1.Start();
       }
```

9.3.5 Methoden für die parallele Ausführung sperren

Neben obigen Sperr-Varianten können Sie auch gleich die ganz große Keule herausholen. Die Rede ist vom Attribut *MethodImpl*, das Sie auf einzelne Methoden anwenden können. In diesem Fall ist die **gesamte Methode** für den Aufruf eines weiteren Threads gesperrt. Sinnvollerweise sollten Sie in dieser Methode nicht allzuviel Code unterbringen, sondern nur die wirklich relevanten Routinen, sonst wird aus Ihrem asynchronen Programm schnell eine ganz normale synchrone Anwendung.

BEISPIEL 9.11: Verwendung des Attributs

C#

```
...
using System.Runtime.CompilerServices;
...
        [MethodImpl(MethodImplOptions.Synchronized)]
        public void Berechne(object anzahl)
        {
            for (int i = 0; i < 5; i++)
            {
                Thread.Sleep(100);
                summe += (int) anzahl;
                Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe);
            }
        }
```

Obige Methode wird in keinem Fall durch andere Threadaufrufe unterbrochen, sondern immer komplett abgearbeitet.

Vergleichsbeispiel

Übertragen auf unser "Bar"-Metapher können Sie sich obiges Beispiel auch als alleiniger Gast in der Bar vorstellen. Es kann immer nur einer eintreten und sich betrinken. Alle anderen warten am Eingang (vor der Methode).

9.3.6 Semaphore

Da wir gerade bei Analogien sind, auch hierzu gleich eine passende:

Vergleichsbeispiel

Vergleichen Sie eine Semaphore mit den Sitzplätzen an der Bar. Sie legen beim Erzeugen einer Semaphore fest, wie viele Sitzplätze es gibt und wie viele davon frei sind. Mit *WaitOne* können Sie prüfen ob ein Platz frei ist und wenn ja, diesen auch gleich belegen. Dieser wird erst beim Verlassen (*Release*) wieder freigegeben. Ist kein Platz frei, blockiert *WaitOne* bis ein Platz frei ist. Alternativ können Sie auch maximale Wartezeiten als Zeitwert übergeben.

Der Nutzen dieses Objekts wird bei begrenzten Ressourcen und einer hohen Anforderungszahl durch verschiedene Threads deutlich.

BEISPIEL 9.12: Die Routine *BerechneMitSemaphore* soll von maximal drei Threads gleichzeitig genutzt werden.

C#

Die Semaphore erzeugen und initialisieren:

```
        private Semaphore sem = new Semaphore(3, 3);  // 3 frei, 3 insgesamt verfügbar
...
```

Wir starten 10 Threads, von denen aber maximal 3 gleichzeitig zum Zuge kommen:

```
        private void button8_Click(object sender, EventArgs e)
        {
            for (int i = 0; i < 10; i++)
            {
                Thread t = new Thread(BerechneMitSemaphore);
                t.Name = "Thread" + i.ToString();
                t.Start();
            }
        }
```

Die eigentliche Routine:

```
        public void BerechneMitSemaphore(object anzahl)
        {
```

Nur wenn noch nicht 3 angemeldete Threads vorliegen, kommen wir hier weiter:

```
            sem.WaitOne();
            for (int i = 0; i < 5; i++)
            {
                Thread.Sleep(100);
                Debug.Print("Thread {0}: ", Thread.CurrentThread.Name);
            }
```

Wir geben die Semaphore (bzw. einen Platz) wieder frei:

```
            sem.Release();
        }
```

Ergebnis

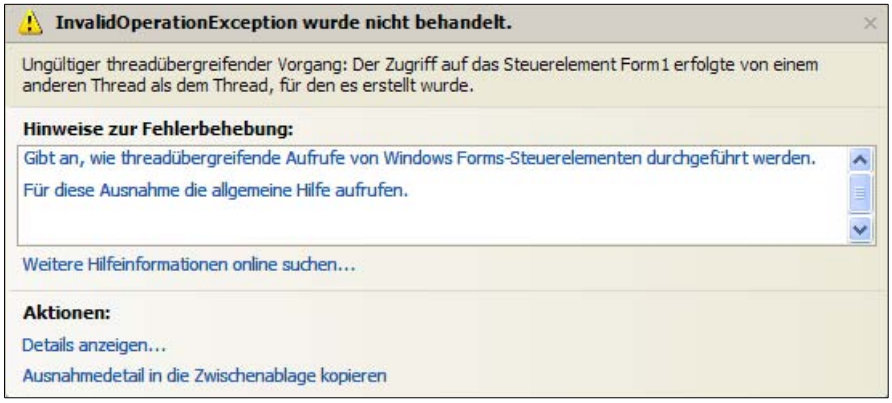
Die Ausgabe: Bei einem Blick ins Ausgabefenster werden Sie feststellen, dass sich maximal 3 Threads gleichzeitig abmühen.

HINWEIS: Natürlich gibt es noch weitere Sperr- und Synchronisationsmechanismen (ReaderWriterLock, Wait-Handles ...) und mehr Methoden als die oben aufgeführten, doch wir wollten hier lediglich einen ersten Einstieg vermitteln. Beachten Sie, dass Sie von den o.g. Sperrmechanismen auch bei der *Task Parallel Library* Gebrauch machen müssen, wenn dies für den Programmablauf erforderlich ist.

Wichtiger ist ein weiteres Problem, mit dem sich der Thread-Programmierer herumschlagen muss: die Interaktion mit dem Vordergrund-Thread, d h., mit unserer Programmoberfläche.

9.4 Interaktion mit der Programmoberfläche

Wie bereits für einige Beispiele angemerkt, sind diese teilweise nicht threadsicher programmiert. Das Problem: Die Steuerelemente eines Windows Forms- bzw. WPF-Programms, d h. deren Eigenschaften und Methoden, sind nicht threadsicher. Wenn wir aus einem anderen Thread als dem erstellenden (Anwendungs-Thread) auf die Komponenten zugreifen (z.B. Aktualisieren einer *ListBox*), kommt es bereits beim ersten Test in Visual Studio zu folgendem unerfreulichen Dialog:



HINWEIS: Starten Sie allerdings das Programm direkt aus dem Explorer heraus, wird die Anwendung mehr oder weniger korrekt ausgeführt, ohne dass eine Fehlermeldung auftritt. Dabei kann es zu unerwünschten Effekten kommen, da die Windows Forms bzw. auch WPF-Controls nicht threadsicher sind.

9.4.1 Die Werkzeuge

Dass der direkte Weg aus einem Thread zum User-Interface versperrt ist, bedeutet noch lange nicht, dass es keine Möglichkeit für eine Interaktion gäbe. Die Lösung heißt in diesem Fall *Invoke* oder *BeginInvoke*. Beide Methoden gestatten es, eine Methode im Kontext des aufrufenden Controls (das kann auch ein Fenster sein) auszuführen. Während *Invoke* synchron ausgeführt wird, d h. auf die Beendigung der aufgerufenen Methode wartet, stellt *BeginInvoke* nur eine neue Anforderung in die Message-Queue des Anwendungs-Threads und unterbricht die Ausführung des aufrufenden Threads nicht.

9.4.2 Einzelne Steuerelemente mit Invoke aktualisieren

Geht es darum, auf die Eigenschaften eines einzelnen Controls zuzugreifen, übergeben Sie einen Delegaten an die *Invoke*-Methode des gewünschten Controls. Diese Methode wird synchron abgearbeitet, d h., die Programmausführung des aktuellen Threads wird hier zunächst gestoppt.

Da die Anzeige in den Steuerelementen ja kaum Selbstzweck ist, müssen in der Regel auch Daten übermittelt werden. Dazu bietet sich die zweite Überladung der *Invoke*-Methode an, der Sie neben

dem Delegaten auch ein Objekt übergeben können. Dieses steht dann in der Callback-Methode zur Verfügung und muss nur noch entsprechend typisiert werden.

BEISPIEL 9.13: In einer *ListBox* sind Ergebnisse aus verschiedenen Threads anzuzeigen.

C#

Definition des Delegaten:

```
public delegate void MyAnzeigeDelegate(string msg);
```

Hier wird der Wert in die *ListBox* eingetragen, dank aufrufender *Invoke*-Methode im Kontext des Vordergrund-Threads:

```
public void myAnzeige(string msg)
{
    listBox1.Items.Add(msg);
}
```

Unsere Callback-Funktion:

```
public void BerechneMitAnzeige(object anzahl)
{
    lock (this)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(200);
            summe++;
            string s = "Thread " + Thread.CurrentThread.Name + ": Summe = " +
            summe.ToString();
```

Zuweisen des Delegaten für die *Invoke*-Methode:

```
            listBox1.Invoke(new MyAnzeigeDelegate(myAnzeige), new object[] { s });
        }
    }
}
```

Der Threadaufruf bzw. -start:

```
private void button5_Click(object sender, EventArgs e)
{
    Thread t1 = new Thread(BerechneMitAnzeige);
    t1.Name = "T" + Id.ToString();
    Id++;
    t1.Start();
}
```

HINWEIS: Alternativ können Sie die Methode auch asynchron mit *BeginInvoke* aufrufen. Warten auf die Abarbeitung können Sie mit *EndInvoke*.

9.4.3 Mehrere Steuerelemente aktualisieren

Wer mehr als ein Control aktualisieren will und die aufwändige Schreiberei mit *Invoke* nicht jedes Mal neu erfinden möchte, der kann auch eine zentrale Anzeige-Routine im Kontext des Formulars erstellen. Da die Steuerelemente im gleichen Kontext laufen, können Sie hier schalten und walten wie Sie es für notwendig halten.

BEISPIEL 9.14: Aktualisieren mehrerer Steuerelemente

C#

```
public delegate void MyAnzeigeDelegate(string msg);

public void myAnzeige(string msg)
{
   listBox1.Items.Add(msg);
   label1.Text = "Bla Bla";
   progressBar1.Value++;
   if (progressBar1.Value == progressBar1.Maximum) progressBar1.Value = 0;

}

public void Anzeige(string msg)
{
   this.Invoke(new MyAnzeigeDelegate(myAnzeige), new object[] { msg });
}
```

Der spätere Aufruf beschränkt sich auf:

```
Form1.Anzeige("Hallo");
```

9.4.4 Ist ein Invoke-Aufruf nötig?

Wer eine universelle Methode schreiben will, die sowohl aus dem Hintergrundthread als auch aus dem Vordergrundthread aufgerufen werden kann, sollte sich die Eigenschaft *InvokeRequired* näher ansehen. Da jedes Control über diese Eigenschaft verfügt, können Sie schnell ermitteln, ob ein *Invoke-Aufruf* erforderlich ist oder nicht.

BEISPIEL 9.15: Verwendung von *InvokeRequired*

C#

```
void MeineAnzeige()
{
    if (listBox1.InvokeRequired)
```

Falls nötig, rufen wir die Methode im Kontext des Controls erneut auf:

```
        listBox1.Invoke(new MethodInvoker(MeineAnzeige));
    else
        listBox1.Items.Add("Hello World!");
}
```

9.4.5 Und was ist mit WPF?

In den bisherigen Ausführungen war immer von den Windows Forms die Rede. Wie bereits erwähnt, sind jedoch auch die WPF-Controls nicht threadsicher, was die Frage aufwirft, wie wir in diesem Fall den Zugriff auf die Oberfläche realisieren.

Die grundsätzliche Vorgehensweise unterscheidet sich nur geringfügig von der Windows Forms-Lösung, statt der Methode *Control.Invoke* verwenden Sie in diesem Fall *Control.**Dispatcher**.Invoke*. Die *InvokeRequired*-Eigenschaft der Windows Forms-Controls wird durch die *CheckAccess*-Methode ersetzt.

HINWEIS: Beachten Sie, dass die *CheckAccess*-Methode *true* zurückliefert, wenn der Aufruf im Vordergrund-Thread erfolgt (die Logik negiert gegenüber *InvokeRequired*).

BEISPIEL 9.16: Verwendung von *Dispatcher.Invoke* in einer WPF-Anwendung

C#

```
...
using System.Threading;
using System.Threading.Tasks;
...
    public partial class MainWindow : Window
    {
        private int id;
```

Definition des Delegaten:

```
        public delegate void MyAnzeigeDelegate(string msg);
```

Hier wird der Wert in die *ListBox* eingetragen, dank aufrufender *Invoke*-Methode im Kontext des Vordergrund-Threads:

```
        public void myAnzeige(string msg)
        {
            listBox1.Items.Add(msg);
        }
```

Unsere Thread-Methode:

```
        public void BerechneMitAnzeige(object anzahl)
        {
            lock (this)
            {
                for (int i = 0; i < 5; i++)
                {
                    Thread.Sleep(200);
                    string s = "Thread " + Thread.CurrentThread.Name + ": Durchlauf = " +
                               i.ToString();
```

Wird die Methode aus dem Vordergrund-Thread aufgerufen, greifen wir direkt auf die *ListBox* zu, andernfalls per *Invoke*:

BEISPIEL 9.16: Verwendung von *Dispatcher.Invoke* in einer WPF-Anwendung

```
                if (listBox1.Dispatcher.CheckAccess())
                    listBox1.Items.Add(s);
                else
                    listBox1.Dispatcher.Invoke(new MyAnzeigeDelegate(myAnzeige),
                                               new object[] { s });
            }
        }
    }
...
```

Thread erzeugen und starten:

```
    private void button1_Click(object sender, RoutedEventArgs e)
    {
        Thread t1 = new Thread(BerechneMitAnzeige);
        t1.Name = "T" + id.ToString();
        id++;
        t1.Start();
    }
```

XAML

Der XAML-Code für die "Oberfläche" (ein *Button* und eine *ListBox*):

```
<Window x:Class="WPF_UI.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">

  <DockPanel  Name="dockPanel1" >
    <Button DockPanel.Dock="Bottom" Content="Button" Height="23" Name="button1"
            Click="button1_Click" />
    <ListBox Name="listBox1" />
  </DockPanel>
</Window>
```

9.5 Timer-Threads

Einen Nachteil hatten die bisherigen Thread-Varianten: Die Methode wird lediglich einmal abgearbeitet, nachfolgend ist der Thread im *Stopped*-Zustand und damit nicht wieder nutzbar. Einige der gezeigten Beispiele versuchten mit Schleifen und der *Sleep*-Anweisung so etwas wie ein zyklisches Verhalten nachzuahmen, dies führt jedoch nur teilweise zum Erfolg und ist auch nicht besonders genau.

Abhilfe verspricht die Klasse *Timer*, die Sie ebenfalls im Namespace *System.Threading* vorfinden. Die übergebene Methode wird in einem Thread aus dem Threadpool ausgeführt.

BEISPIEL 9.17: Verwendung eines Timer-Threads

C#

```
...
using System.Threading;
namespace ThreadLocking
{
    public partial class Form1 : Form
    {
        private System.Threading.Timer myTimer;
```

Beim Instanziieren der *Timer*-Klasse übergeben Sie folgende Parameter: den Delegaten, eventuell ein Objekt, das als Parameter für die Timer-Callback-Methode verwendet werden kann (sonst *null*), die Wartezeit bis zum Start des Timers (*Timeout.Infinite* um den sofortigen Start zu verhindern) sowie das Zeitintervall in Millisekunden (oder als *TimeSpan*).

```
        private void button6_Click(object sender, EventArgs e)
        {
            myTimer = new System.Threading.Timer(MeinTimer, "Timer" , 1000, 500);
        }
```

Die Callback-Routine (mit dem Parameter-Objekt):

```
        public void MeinTimer(object obj)
        {
            string s = (string)obj;
            listBox1.Invoke(new MyAnzeigeDelegate(myAnzeige), new object[] { s });
        }
```

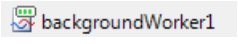
Anzeige in der *ListBox*:

```
        public delegate void MyAnzeigeDelegate(string msg);

        public void myAnzeige(string msg)
        {
            listBox1.Items.Add(msg);
        }
```

HINWEIS: Mit der *Change*-Methode können Sie Startzeit und Intervall nachträglich ändern!

9.6 Die BackgroundWorker-Komponente

Fast hätten wir es vergessen, im Werkzeugkasten findet sich auch noch etwas, was im Zusammenhang mit der asynchronen Programmausführung von Interesse sein könnte. Die Rede ist vom *BackgroundWorker*-Steuerelement:



Dieses Control ermöglicht es Ihnen, die Routineaufgaben, die im Zusammenhang mit einem Hintergrund-Thread anfallen, in einem vorgegebenen Grundgerüst zu programmieren:

- Bereitstellen eines Threads
- Übergabe und Rückgabe von Objekten
- Fortschrittsmeldung im Vordergrund-Thread
- Abbruchmöglichkeit
- Fehlerabfrage

Dass bei dieser Vielfalt die Programmierung nicht mit zwei Zeilen abgetan ist, ahnen Sie bereits. Ein Beispiel zeigt die Verwendung.

BEISPIEL 9.18: Hintergrundberechnung mit Hilfe eines *BackgroundWorker*-Steuerelements

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;
using System.Diagnostics;

namespace BackgroundWorkerTest
{
    public partial class Form1 : Form
    {
```

Die Hintergrund-Routine:

```
        private void backgroundWorker1_DoWork(object sender, DoWorkEventArgs e)
        {
            Int32 res = 0;
```

Über *e.Argument* können Sie den Startparameter abfragen. Sie können beliebige Objekte verwenden, müssen diese jedoch typisieren:

```
            for (int i = 0; i < (int)e.Argument; i++)
            {
```

Hat der Nutzer einen Abbruch angefordert (*CancelAsync*-Methode), können Sie dies über *CancellationPending* auswerten:

```
                if ((sender as BackgroundWorker).CancellationPending)
                {
                    e.Cancel = true;
                    break;
                }
                res += i;
```

Hier produzieren wir etwas Luft:

```
                Thread.Sleep(100);
```

BEISPIEL 9.18: Hintergrundberechnung mit Hilfe eines *BackgroundWorker*-Steuerelements

```
            Debug.Print("BackgroundWorker: " + res.ToString());
```

Soll ein Fortschritt angezeigt werden (*WorkerReportsProgress=true*), können Sie hier den Wert übergeben:

```
            (sender as BackgroundWorker).ReportProgress((int)((float)i /
                                    (float)(int)e.Argument * 100));
        }
```

Das Ergebnis lässt sich per *e.Result* zurückgeben, auch hier sind beliebige Objekte möglich:

```
        e.Result = res;
    }
```

In obiger Ereignisprozedur ist kein Zugriff auf die Oberfläche zulässig, verwenden Sie dafür das *ProgressChanged*-Ereignis!

Den Fortschritt Ihrer Bemühungen können Sie im folgenden Ereignis auswerten und anzeigen lassen:

```
    private void backgroundWorker1_ProgressChanged(object sender,
                                                   ProgressChangedEventArgs e)
    {
        progressBar1.Value = e.ProgressPercentage;
    }
```

Wurde der Thread abgearbeitet oder findet ein Nutzerabbruch statt, wird das folgende Ereignis ausgeführt:

```
    private void backgroundWorker1_RunWorkerCompleted(object sender,
                                                      RunWorkerCompletedEventArgs e)
    {
```

Falls ein Fehler aufgetreten ist:

```
        if (e.Error != null)
            MessageBox.Show(e.Error.Message);
```

Oder es handelt sich um einen Nutzerabbruch:

```
        else if (e.Cancelled)
           listBox1.Items.Add("Nutzerabbruch");
```

Oder es ist alles in Ordnung und wir können das Ergebnis auswerten:

```
        else
          listBox1.Items.Add("ERGEBNIS:" + e.Result.ToString());
        button1.Enabled = true;
        button2.Enabled = false;
        progressBar1.Value = 0;
    }
```

BEISPIEL 9.18: Hintergrundberechnung mit Hilfe eines *BackgroundWorker*-Steuerelements

Last but not least, müssen wir den Thread auch noch starten:

```
        private void button1_Click(object sender, EventArgs e)
        {
            button1.Enabled = false;
            button2.Enabled = true;
            backgroundWorker1.RunWorkerAsync(100);
        }
```

Wichtig: Hier können Sie ein Objekt als Parameter an den Thread übergeben.

Ist Ihnen nach Abbruch zumute, hilft die Methode *CancelAsync*:

```
        private void button2_Click(object sender, EventArgs e)
        {
            backgroundWorker1.CancelAsync();
            button2.Enabled = false;
        }
    }
}
```

Wie Sie sehen, ganz trivial ist die Verwendung der Komponente nicht, dafür erhalten Sie aber gleich einen recht umfangreichen Rahmen, der fast alle Eventualitäten abdeckt.

9.7 Asynchrone Programmier-Entwurfsmuster

Nachdem wir uns mit den Grundlagen der Thread-Programmierung vertraut gemacht haben, wollen wir einen Blick auf weitere Möglichkeiten asynchroner Programmierung mittels Delegates werfen. Einerseits bietet das .NET-Framework bereits einige Klassen mit integriertem Multithreading, andererseits können Sie auch selbst beliebige Methoden per Delegate asynchron aufrufen (mehr dazu im Abschnitt 9.8).

9.7.1 Kurzübersicht

Mit dem asynchronen Programmierentwurfsmuster (*Asynchronous Design Pattern*) bietet sich dem Nutzer diverser .NET-Klassen die Entscheidungsmöglichkeit, ob eine Methode synchron oder asynchron aufgerufen werden soll.

Dazu werden neben der synchronen Methode (z.B. *Read*) noch zwei zusätzliche Methoden mit der Bezeichnung ***Begin***<*MethodenName*> und ***End***<*Methodenname*> eingeführt.

Begin...-Methode

Die ***Begin***<*MethodenName*>-Methode gibt dem Aufrufer ein Objekt vom Typ *IAsyncResult* zurück, über das der Status der jetzt im Hintergrund laufenden Operation ausgewertet werden kann (zum Beispiel durch Polling).

An die Methode übergeben Sie zunächst ebenfalls die gleichen Parameter wie für das synchrone Pendant, zusätzlich werden noch ein Rückrufdelegate vom Typ *AsyncCallback* und ein beliebiges Objekt für die Datenübergabe an den Thread übergeben.

***BEISPIEL 9.19: BeginRead*-Methode**

C#
```
IAsyncResult res = stateobj.fs.BeginRead(stateobj.Bytes, 0,(int) stateobj.fs.Length,
                                         Fertig, stateobj);
```

Fertig ist der Rückrufdelegate, *stateobj* ist ein nutzerdefiniertes Objekt.

End...-Methode

Diese können Sie in der Rückruf-Methode aufrufen, um das Endergebnis des asynchronen Methodenaufrufs zu ermitteln.

Alternativ kann die Methode auch nach dem Eintreten der Bedingung *IsCompleted* aufgerufen werden.

***BEISPIEL 9.20: EndRead*-Methode**

C#
```
IAsyncResult res  = fs.BeginRead(Bytes, 0, (int)fs.Length, null, null);
while (!res.IsCompleted)
  int BytesGelesen = fs.EndRead(res);
```

Die Rückrufmethode

Die Rückrufmethode wird nach dem Ende der asynchronen Methode aufgerufen. Sie muss folgendem Muster entsprechen:

```
public void <Methodenname>(IAsyncResult asyncResult)
{
```

Über *asyncResult.AsyncState* können Sie auf das eventuell übergebene Nutzer-Objekt zugreifen:

```
    myState state = (myState)asyncResult.AsyncState;
```

In dieser Methode rufen Sie im Allgemeinen das Funktionsergebnis per *End<Methodenname>* ab. Dafür benötigen Sie den Paremeter *asyncResult*:

```
state.BytesGelesen = state.fs.EndRead(asyncResult);
```

Wie Sie das Entwurfsmuster konkret einsetzen, sollen Ihnen die folgenden Beispiele demonstrieren.

9.7.2 Polling

Es wurde bereits erwähnt, dass eine der Möglichkeiten, auf das Endergebnis der asynchronen Operation zu warten, die Statusabfrage der Operation per Polling ist. Dazu wird der Rückgabewert (Typ *IAsyncResult*) der jeweiligen *Begin...*-Methode ausgewertet, dessen Eigenschaft *IsCompleted* gibt über den aktuellen Zustand der asynchronen Operation Auskunft. Die Abfrage kann per

Schleife oder auch zu festgelegten Zeitpunkten mit einem *Timer* erfolgen. Ein kleines Beispiel zeigt die Vorgehensweise.

BEISPIEL 9.21: Polling

C#

Eine große Datei soll asynchron in ein Byte-Array geladen werden. Abschließend ist die Anzahl der gelesenen Bytes zu bestimmen[1].

Den erforderlichen Namespace einbinden:

```
...
using System.IO;

namespace ASync1
{
    public partial class Form1 : Form
    {
...
        private void button1_Click(object sender, EventArgs e)
        {
            listBox1.Items.Add("Starte Lese-Vorgang ...");
```

Einen *FileStream* erzeugen:

```
            FileStream fs = new FileStream("01.jpg", FileMode.Open, FileAccess.Read,
                                           FileShare.Read, 1000, true);
```

Das Byte-Array anlegen:

```
            byte[] Bytes = new byte[fs.Length];
```

Den asynchronen Aufruf starten:

```
            IAsyncResult res  = fs.BeginRead(Bytes, 0, (int)fs.Length, null, null);
```

Der Rückgabewert ist vom Typ *IAsyncResult*. Mit einer Schleife fragen wir den Status solange ab, bis die Operation abgeschlossen ist[2]:

```
            while (!res.IsCompleted)
                listBox1.Items.Add("IsCompleted = " + res.IsCompleted.ToString());
            listBox1.Items.Add("IsCompleted = " + res.IsCompleted.ToString());
```

Bestimmen der Anzahl gelesener Bytes:

```
            int BytesGelesen = fs.EndRead(res);
            listBox1.Items.Add("Gelesen: " +  BytesGelesen.ToString() + " Bytes");
        }
    }
}
```

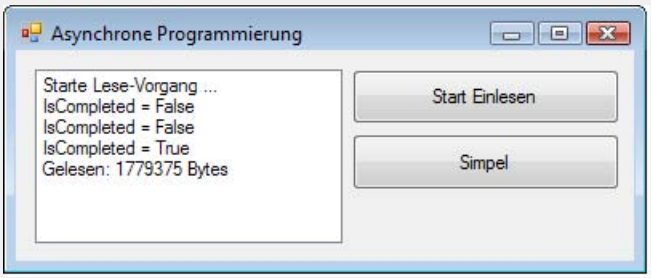
[1] Das Beispiel scheint zu diesem Zeitpunkt noch ziemlich sinnlos zu sein, wird aber als Vorlage für die folgenden Beispiele gebraucht.

[2] Mit einem synchronen Dateizugriff hätten wir mit weniger Aufwand dasselbe erreicht, aber es geht hier ja um asynchrone Aufrufe und die Abfrage könnte ja auch per Timer etc. erfolgen.

BEISPIEL 9.21: Polling

Ergebnis

Ein Start des Programms zeigt, was passiert:



Nach dem Start der *BeginRead*-Methode wird in diesem Fall die *while*-Schleife zweimal durchlaufen. Danach können wir die Anzahl der gelesenen Bytes bestimmen.

Wie oft die *while*-Schleife durchlaufen wird, hängt von der Zugriffsgeschwindigkeit auf den *FileStream* ab. Es könnten also auch mehr oder weniger Durchläufe notwendig sein.

Wie es besser (und sinnvoller) geht, zeigt der nächste Abschnitt.

9.7.3 Callback verwenden

Mit einer Callback-Methode kann die asynchrone Operation ohne Unterbrechung durch den aufrufenden Thread (Polling) durchgeführt werden. Ist die Operation zu Ende, ruft sie die Callback-Methode auf.

HINWEIS: Die Callback-Methode muss einem bestimmten Muster entsprechen, als Parameter wird ein *IAsyncResult*-Objekt erwartet.

BEISPIEL 9.22: Callback

C#

Wir schreiben das Beispiel aus dem vorhergehenden Abschnitt so um, dass wir per Callback auf das Ende reagieren können.

```
        public void Fertig(IAsyncResult asyncResult)
        {
         // hier ist die Operation zu Ende
        }


...
       fs.BeginRead(Bytes, 0, (int)fs.Length, Fertig, null);
...
```

Folgende Einschränkungen sind allerdings zu beachten:

- Da die Methode *Fertig* in einem anderen Thread als die aufrufende Routine läuft, können wir nicht sicher auf deren Eigenschaften zugreifen.
- Auch der Zugriff auf die Steuerelemente des Formulars (in diesem Fall die *ListBox*) bleibt uns aus diesem Grund verwehrt.

Deshalb wagen wir einen zweiten Anlauf.

9.7.4 Callback mit Parameterübergabe verwenden

Die Übergabe der notwendigen Variablen/Eigenschaften an die Callback-Routine lässt sich mit Hilfe einer eigenen Klasse realisieren. Eine Instanz dieser Klasse kann als Parameter an die *Begin...*-Methode übergeben werden.

In der Callback-Routine stellt der *IAsyncResult*-Parameter mit der Eigenschaft *AsyncState* das übergebene Objekt wieder zur Verfügung.

HINWEIS: Sie müssen das Objekt erst typisieren!

BEISPIEL 9.23: Callback mit Parameterübergabe

C#

```
...
    public partial class Form1 : Form
    {
```

Die Hilfsklasse:

```
        public class myState
        {
            public byte[] Bytes;
            public FileStream fs;
            public int BytesGelesen;
        }
```

Die neue Callback-Routine:

```
        public void Fertig(IAsyncResult asyncResult)
        {
```

Hilfsobjekt typisieren und verwenden:

```
            myState state = (myState)asyncResult.AsyncState;
            state.BytesGelesen = state.fs.EndRead(asyncResult);
            state.fs.Close();
        }
```

Das Starten des asynchronen Dateizugriffs:

```
        private void button1_Click(object sender, EventArgs e)
        {
```

BEISPIEL 9.23: Callback mit Parameterübergabe

```
            listBox1.Items.Add("Starte Lese-Vorgang ...");
```

Hier erzeugen wir unser Hilfsobjekt:

```
            myState stateobj = new myState();
```

Die erforderlichen Daten speichern:

```
            stateobj.fs = new FileStream("01.jpg", FileMode.Open, FileAccess.Read,
                                         FileShare.Read, 1000, true);
            stateobj.Bytes = new byte[stateobj.fs.Length];
```

Asynchronen Aufruf mit Hilfsobjekt als Parameter starten:

```
            IAsyncResult res = stateobj.fs.BeginRead(stateobj.Bytes, 0,(int)
                               stateobj.fs.Length, Fertig, stateobj);
        }
    }
}
```

Das sieht doch schon wesentlich besser aus als im letzten Beispiel! In der Callback-Methode können die geschriebenen Bytes bestimmt und der *FileStream* geschlossen werden.

Doch etwas fehlt noch: die Interaktion mit der Programmoberfläche!

9.7.5 Callback mit Zugriff auf die Programm-Oberfläche

Um aus dem Callback-Thread auf die Programmoberfläche (Main-Thread) zugreifen zu können, müssen wir per *Invoke* oder *BeginInvoke* mittels Delegate die gewünschte Methode des Formulars aufrufen.

BEISPIEL 9.24: Zugriff auf die Programmoberfläche

C#

```
...
    public partial class Form1 : Form
    {
```

Unsere Routine, die auf die Steuerelemente zugreifen kann:

```
        public void FormRefresh(myState obj)
        {
            listBox1.Items.Add("CallBack: --- Fertig ---");
            listBox1.Items.Add("CallBack: Gelesen: " + obj.BytesGelesen.ToString() +
                               "Bytes");
        }
```

Wie Sie sehen, übergeben wir auch unser Hilfsobjekt gleich noch als Parameter, so können wir bequem mit dessen Informationen arbeiten.

Wir deklarieren einen Delegate für den Aufruf der *Invoke*-Methode:

```
        private delegate void FormRefreshDelegate(myState obj);
```

BEISPIEL 9.24: Zugriff auf die Programmoberfläche

Und zum Schluss ändern wir noch die Callback-Routine:

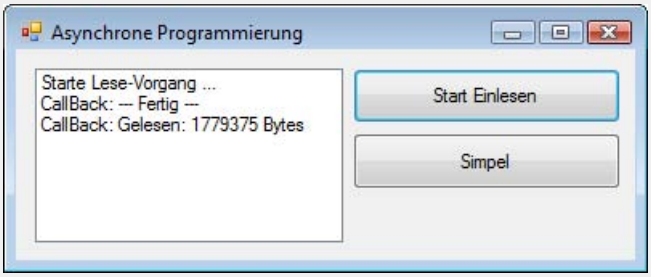
```
public void Fertig(IAsyncResult asyncResult)
{
    myState state = (myState)asyncResult.AsyncState;
    state.BytesGelesen = state.fs.EndRead(asyncResult);
```

Übergabe von Delegate und Hilfsobjekt:

```
    this.Invoke(new FormRefreshDelegate(FormRefresh), state);
    state.fs.Close();
}
```

Ergebnis

Jetzt funktioniert auch unser Programm mittels Callback-Routine:



Egal wie lange der Einlesevorgang für die Datei ist, die Oberfläche bleibt bedienbar. Ist die Operation abgeschlossen, sorgt die Callback-Routine für die Anzeige der Werte.

HINWEIS: Die *Invoke*-Methode wird synchron aufgerufen, d h., erst wenn der Thread des Fensters abgearbeitet ist, wird die Programmausführung fortgesetzt.

9.8 Asynchroner Aufruf beliebiger Methoden

Möchten Sie nicht nur die vom .NET-Framework bereitgestellten asynchronen Methoden aufrufen, sondern auch beliebige Methoden eigener Klassen parallel verarbeiten? Wenn ja, dann sind Sie hier richtig.

9.8.1 Die Beispielklasse

Um Sie nicht mit trockener Theorie zu langweilen, fangen wir lieber gleich mit einer übersichtlichen Beispielklasse an, die zunächst eine synchrone Methode *Berechne* zur Verfügung stellen soll:

BEISPIEL 9.25: Beispielklasse definieren

C#

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Media;

namespace Async2
{
```

Die Klassendefinition:

```
    class LangeBerechnung
    {
```

Eine Eigenschaft für das Endergebnis der "Berechnung":

```
        public int Ergebnis;
```

Die eigentliche Funktion[1]:

```
        public int Berechne(int anzahl)        {
            Ergebnis = 0;
            for (int i = 0; i < anzahl; i++)
            {
```

Wir produzieren etwas Laufzeit und eine akustische Rückmeldung:

```
                Thread.Sleep(1000);
                SystemSounds.Beep.Play();
                Ergebnis += i;
            }
            return Ergebnis;
        }
    }
```

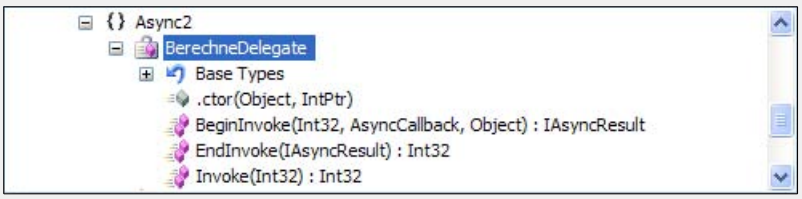
Grundlage für den asynchronen Aufruf einer Methode ist die Definition und Instanziierung eines entsprechenden Delegaten! Deshalb an dieser Stelle:

```
    public delegate int BerechneDelegate(int anzahl);
}
```

Der Delegate muss mit der Definition der aufzurufenden Methode übereinstimmen. Der Compiler erzeugt uns für diesen Delegate automatisch die drei Methoden *BeginInvoke*, *Invoke* und *EndInvoke*, wie ein Blick in die spätere Assembly zeigt:

[1] Die Berechnungsfunktion soll hier nicht im Mittelpunkt stehen ...

BEISPIEL 9.25: Beispielklasse definieren



Beachten Sie auch die *EndInvoke*-Methode, diese gibt das Funktionsergebnis zurück.

HINWEIS: Wie Sie obige Methode synchron aufrufen, brauchen wir Ihnen sicher nicht zu demonstrieren.

9.8.2 Asynchroner Aufruf ohne Callback

Im ersten Schritt wollen wir uns zunächst von der asynchronen Funktionsweise überzeugen, bevor wir mit Callbacks und Userinterface arbeiten.

BEISPIEL 9.26: Fortsetzung des Vorgängerbeispiels

C#

Mit dem Button-Klick wird folgender Code abgearbeitet

```
listBox1.Items.Clear();
listBox1.Items.Add("Start");
```

Instanz bilden:

```
LangeBerechnung lb = new LangeBerechnung();
```

Delegate instanziieren und Methode zuweisen:

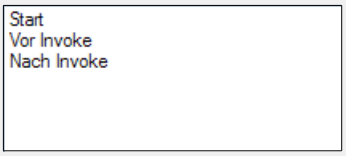
```
BerechneDelegate dg = new BerechneDelegate(lb.Berechne);
listBox1.Items.Add("Vor Invoke");
```

Hier erfolgt der asynchrone Aufruf, die *BeginInvoke*-Methode hat der Compiler für uns erzeugt:

```
dg.BeginInvoke(5, null, null);
listBox1.Items.Add("Nach Invoke");
```

Ergebnis

Ein Test lässt uns lediglich anhand der *ListBox*-Einträge (diese werden ohne Unterbrechung sofort ausgegeben) ...



BEISPIEL 9.26: Fortsetzung des Vorgängerbeispiels

... und den Tönen aus dem "Untergrund" ahnen, dass die gewünschte Funktionalität (der Vordergrund-Thread läuft ungestört weiter) realisiert worden ist.

9.8.3 Asynchroner Aufruf mit Callback und Anzeigefunktion

BEISPIEL 9.27: Modifizieren des Vorgängerbeispiels durch Einfügen von Callback- und Anzeige-Funktion

C#

Wir deklarieren zunächst eine Callback-Funktion (Parameter nicht vergessen):

```
private void MyCallback(IAsyncResult res)
{
    this.Invoke(new anzeigedelegate(anzeige), new object[] { "Callback: Fertig" });
}
```

Wie Sie sehen, müssen wir wieder etwas Aufwand treiben, um Bildschirmausgaben zu produzieren (per *Invoke* wechseln wir in den Vordergrundthread):

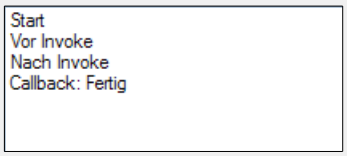
```
public delegate void anzeigedelegate(string msg);

public void anzeige(string msg)
{
   listBox1.Items.Add(msg);
}
```

Unsere Vorbereitungen für den Aufruf der *BeginInvoke*-Methode sind fast gleich geblieben:

```
...
    LangeBerechnung lb = new LangeBerechnung();
    BerechneDelegate dg = new BerechneDelegate(lb.Berechne);
    listBox1.Items.Add("Vor Invoke");
    dg.BeginInvoke(5, MyCallback, null);
...
```

Ergebnis

Ein Test bringt jetzt endlich auch etwas auf den Bildschirm:



Die fünf Töne, die ausgegeben werden, sollten Sie von der Funktionsfähigkeit überzeugen.

9.8.4 Aufruf mit Rückgabewerten (per Eigenschaft)

Doch etwas fehlt noch: Die Funktion hat ja keinen Selbstzweck sondern soll auch ein Berechnungsergebnis zurückgeben. Zwei Varianten bieten sich an:

- Entweder Sie verwenden Eigenschaften der Berechnungsklasse (wie in unserem Beispiel (*Ergebnis*) bzw. durch Definition einer kapselnden Hilfsklasse).
- Oder Sie nutzen die Möglichkeit per *EndInvoke* den Rückgabewert zu ermitteln.

Wir stellen Ihnen zunächst die erste Variante am Beispiel vor.

BEISPIEL 9.28: Rückgabewert per Eigenschaft

C#

Anpassen der Callback-Routine:

```
private void MyCallback(IAsyncResult res)
{
   if (res.AsyncState != null)
   {
      this.Invoke(new anzeigedelegate(anzeige), new object[] { "Callback: Fertig" });
```

Rückgabewert per Eigenschaft:

```
      this.Invoke(new anzeigedelegate(anzeige), new object[] { "Ergebnis" +
                  (res.AsyncState as LangeBerechnung).Ergebnis.ToString() });
   } else
      this.Invoke(new anzeigedelegate(anzeige), new object[] { "Callback: Fertig" });
}
```

Im Beispiel wird *res.AsyncState* als Objekt vom Typ *LangeBerechnung* typisiert und abgefragt. Natürlich müssen wir jetzt auch das erforderliche Objekt beim *BeginInvoke*-Aufruf mit übergeben:

```
LangeBerechnung lb = new LangeBerechnung();
BerechneDelegate dg = new BerechneDelegate(lb.Berechne);
listBox1.Items.Add("Vor Invoke");
dg.BeginInvoke(5, MyCallback, lb);
```

Unser Objekt *lb* wird am Ende des Methodenaufrufs an die Callback-Routine im Parameter *res* durchgereicht und kann nach der Typisierung entsprechend ausgewertet werden.

9.8.5 Aufruf mit Rückgabewerten (per EndInvoke)

Wie schon erwähnt, stellt das Framework mit der *EndInvoke*-Methode einen Weg zur Verfügung, direkt an das Funktionsergebnis des asynchronen Methodenaufrufs zu gelangen.

Allerdings müssen Sie in diesem Fall den Delegaten an die *BeginInvoke*-Methode übergeben und auch die Callback-Routine entsprechend anpassen.

BEISPIEL 9.29: Rückgabewert per *EndInvoke*

C#
```
LangeBerechnung lb = new LangeBerechnung();
BerechneDelegate dg = new BerechneDelegate(lb.Berechne);
listBox1.Items.Add("Vor Invoke");
dg.BeginInvoke(5, MyCallback2, dg);
```

Die Callback-Methode:

```
private void MyCallback2(IAsyncResult res)
{
   if (res.AsyncState != null)
   {
      this.Invoke(new anzeigedelegate(anzeige), new object[] { "Callback: Fertig" });
      int Result = ((BerechneDelegate)res.AsyncState).EndInvoke(res);
      this.Invoke(new anzeigedelegate(anzeige), new object[] {"Ergebnis" +
          Result.ToString()});
   }
}
```

HINWEIS: Welche der beiden Varianten zur Auswertung des Funktionsergebnisses Sie bevorzugen bleibt Ihnen überlassen, das Ergebnis ist dasselbe!

9.9 Es geht auch einfacher – async und await

Sicher ist es Ihnen nicht verborgen geblieben, dass es teilweise schon richtig Arbeit macht, wenn Sie von den Vorteilen der asynchronen Programmierung profitieren wollen. Schnell verlieren Sie den Überblick in einem Gestrüpp aus Threads und Delegates und zum Schluss verliert mancher auch die Lust am asynchronen Programmieren.

Dass ist auch den Microsoft-Entwicklern nicht verborgen geblieben, und so führt C# 5.0 im Zusammenhang mit der asynchronen Programmierung zwei neue Schlüsselwörter ein[1]:

- den *async*-Modifizierer und
- den *await*-Operator.

Eine mit *async* markierte Methode wird als "asynchrone Methode" bezeichnet, d.h. die Programmausführung wartet nicht nach deren Aufruf auf das Ende der Methode sondern setzt die Ausführung fort. Mit dem *await*-Operator haben Sie die Möglichkeit, auf das Ende der Methode zu warten und gleichzeitig können Sie den Rückgabewert der Methode auswerten.

HINWEIS: Die neue Windows Runtime (siehe dazu ab Kapitel 17) treibt dass Ganze auf die Spitze. Um die Oberfläche von Rucklern und Wartezeiten zu befreien, sind alle WinRT-Aufrufe mit eine Laufzeit von mehr als 50ms asynchron programmiert.

[1] Das ist eigentlich nicht ganz richtig, unter der Adresse *http //www.microsoft.com/en-us/download/details.aspx?id=9983* finden Sie das *Visual Studio Async CTP* auch für Visual Studio 2010.

Die sich aus obigen Neuerungen ergebenden Erleichterungen für den Programmierer wollen wir zunächst anhand zweier Beispiele demonstrieren, in welchen wir den synchronen mit dem asynchrone Dateizugriff vergleichen.

In einem zweiten Schritt sehen wir uns an, was man dabei alles falsch machen kann, bevor wir abschließend einen Blick darauf werfen, wie Sie Ihre eigenen Routinen in asynchrone Methoden verwandeln können.

9.9.1 Der Weg von Synchron zu Asynchron

Stellen Sie sich vor, Ihre Anwendung soll viele Daten speichern. Der "Flaschenhals" ist, wie nicht anders zu erwarten, die Festplatte mit ihrem begrenzten Datendurchsatz.

BEISPIEL 9.30: Synchrones Speichern von Daten

C#

```
...
using System.IO;
...
```

Unsere Dummy-Daten zum Speichern (500 KB-String):

```
        string dummy = "#".PadLeft(5000000, '#');
...
        private void Button_Click_7(object sender, RoutedEventArgs e)
        {
            listBox1.Items.Clear();
            listBox1.Items.Insert(0, "Start");
```

Datei erzeugen:

```
            using (StreamWriter writer = File.CreateText("VieleDaten.txt"))
            {
                for (int i = 0; i < 1000; i++)
                {
```

Wir speichern den Dummy-String mehrfach mit der *Write*-Methode:

```
                    writer.Write(System.DateTime.Now.ToString() + dummy);
                    listBox1.Items.Insert(0, "Block " + i.ToString());
                }
            }
            listBox1.Items.Insert(0, "FERTIG");
```

Datei wieder löschen, sonst ist Ihre Festplatte bald voll:

```
            File.Delete("VieleDaten.txt");
        }
...
```

HINWEIS: Haben Sie einen langsamen Computer empfehlen wir Ihnen die obigen Schleifen etwas zu kürzen. Besitzer einer SSD sollten die Werte erhöhen.

Nach dem Klick auf die Schaltfläche ist das Formular "eingefroren", eine Bedienung ist nicht möglich, da die Routine im Vordergrundthread läuft und die Kontrolle nicht abgegeben wird. Auch die Anzeige der Zwischenschritte in der *ListBox* erfolgt nicht.

Obiges Beispiel werden wir nun so umschreiben, dass der Benutzer auch während der Ausführung mit dem Formular arbeiten kann.

BEISPIEL 9.31: Asynchrones Speichern

C#

...

Wir markieren unsere Methode als asynchron:

```
private async void Button_Click_8(object sender, RoutedEventArgs e)
{
    listBox1.Items.Clear();
    listBox1.Items.Insert(0, "Start");
    using (StreamWriter writer = File.CreateText("VieleDaten.txt"))
    {
        for (int i = 0; i < 1000; i++)
        {
```

Wir verwenden die asynchrone Variante der *Write*-Methode und warten an dieser Stelle auf das Ende der Ausführung:

```
            await writer.WriteAsync(System.DateTime.Now.ToString() + dummy);
            listBox1.Items.Insert(0, "Block " + i.ToString());
        }
    }
    listBox1.Items.Insert(0, "FERTIG");
    File.Delete("VieleDaten.txt");
}
```

Im Unterschied zur synchronen Variante wird ein "Einfrieren" der Anwendung gänzlich vermieden, d.h., Sie können auch direkt nach dem Klick auf die Schaltfläche das Formular verschieben, Eingaben vornehmen etc.

HINWEIS: Wie Sie sehen, halten sich die Änderungen gegenüber dem letzten Beispiel in Grenzen, eine Threading-Variante ist viel aufwändiger, zumal Sie bei obigem Beispiel jederzeit die Ausgaben im Formular nachverfolgen können. Bei Verwendung eines Threads müssten Sie erst mühevoll in den Vordergrundthread umschalten.

Sehen wir uns obigen Beispiel noch einmal genauer an.

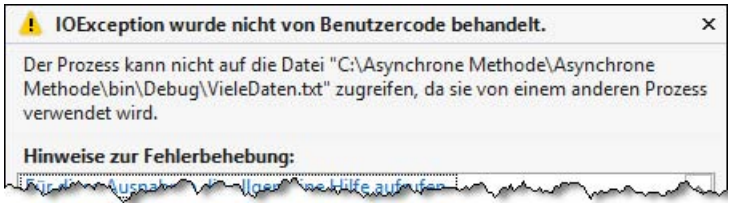
Die gesamte Methode *Button_Click_8* wird im Vordergrund-Thread ausgeführt (vorteilhaft für die uneingeschränkte Interaktion mit der Userinterface), der *WriteAsync*-Aufruf läuft hingegen in einem extra Thread und gibt damit Rechenzeit für den Vordergrund und damit für das Userinterface frei. Der *await*-Operator ermöglicht uns eine synchrone Verarbeitung innerhalb der asynchronen Methode.

Doch was passiert eigentlich, wenn wir auf den *await*-Operator verzichten?

In diesem Fall wird die gesamte Methode *Button_Click_8* synchron ausgeführt, der Methodenaufruf *WriteAsync* jedoch asynchron. Es kommt zum Laufzeitfehler, da die Schleife die Methode mehrfach aufruft und damit ein mehrfacher Dateizugriff erfolgt.

9.9.2 Achtung: Fehlerquellen!

Ja, es könnte so schön sein, einfach alles in einer Methode kapseln und diese dann asynchron aufrufen. Auf obiges Beispiel übertragen ist jedoch eine potentielle Fehlerquelle ganz schnell feststellbar. Klicken Sie doch während des Speicherns der Datei noch einmal auf die Schaltfläche. Die Folge ist ein zweiter, paralleler Aufruf. In diesem Fall kommt es zu einem IO-Fehler, da die Datei durch den ersten Aufruf noch gesperrt ist:



Sie müssen also Sorge dafür tragen, dass in diesem Fall die Schaltfläche nicht mehrfach gedrückt werden kann (*IsEnabled=False*).

Ganz ähnlich stellt sich der Fall dar, wenn Sie ein "sauberes" Programm schreiben wollen, und dazu die Routine in eine extra Methode auslagern:

BEISPIEL 9.32: Das gibt Ärger

C#

Der ausgelagerte, asynchrone Code in einer extra Routine:

```
...
        async private void DateiSpeichern()
        {
            using (StreamWriter writer = File.CreateText("VieleDaten.txt"))
            {
                for (int i = 0; i < 1000; i++)
                {
                    await writer.WriteAsync(System.DateTime.Now.ToString() + dummy);
                    listBox1.Items.Insert(0, "Block " + i.ToString());
                }
            }
        }
```

Der Aufruf obiger Methode:

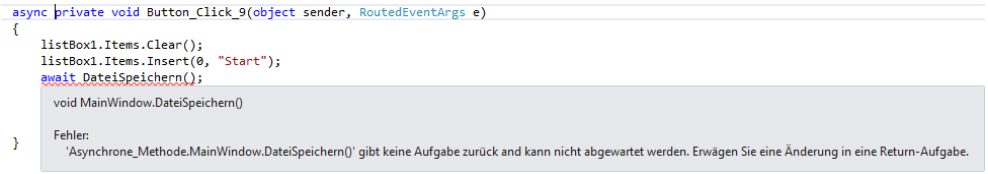
```
        private void Button_Click_9(object sender, RoutedEventArgs e)
        {
            listBox1.Items.Clear();
```

BEISPIEL 9.32: Das gibt Ärger

```
            listBox1.Items.Insert(0, "Start");
            DateiSpeichern();
            listBox1.Items.Insert(0, "FERTIG");
            File.Delete("VieleDaten.txt");
        }
...
```

Raten Sie mal, was hier falsch läuft? Der IO-Fehler wird bei *File.Delete* auftreten, der logische Fehler bereits eine Zeile höher, da hier ein "Fertig" angezeigt wird, obwohl die Routine *DateiSpeichern* noch läuft. Wie stand weiter oben: es handelt sich um eine **asynchrone** Routine (*async*)!

Wer jetzt an den *await*-Operator denkt, liegt nicht ganz falsch. Aber leider, leider wird das so nichts:



Unsere asynchrone Methode muss schon einer speziellen Konvention folgen, um den *await*-Operator nutzen zu können. Die Methode muss in diesen Fall ein *Thread* oder ein *Thread<>* zurückgeben.

BEISPIEL 9.33: Diese Methode kann mit *await* genutzt werden

C#

```
...
    async private Task DateiSpeichern2()
       {
           using (StreamWriter writer = File.CreateText("VieleDaten.txt"))
           {
               for (int i = 0; i < 1000; i++)
               {
                   await writer.WriteAsync(System.DateTime.Now.ToString() + dummy);
                   listBox1.Items.Insert(0, "Block " + i.ToString());
               }
           }
       }
```

Jetzt lässt sich die Methode korrekt verwenden:

```
...
        await DateiSpeichern2();
...
```

Um den Rückgabewert, d h. den Thread, kümmert sich in diesem Fall der Compiler.

9.9.3 Eigene asynchrone Methoden entwickeln

Sicher nicht ganz uninteressant ist die Frage, wie Sie Ihren eigen Code so umschreiben/konvertieren können, dass Sie oder auch Anwender Ihrer Libraries die *async*-Features nutzen können.

Ausgangspunkt soll folgende bisherige Routine sein:

BEISPIEL 9.34: Alter, synchroner Code

C#

```
...
        private int MeineLangeBerechnung()
        {
            int res = 0;
            for (int i = 0; i < 50; i++)
            {
                res+=i;
```

Zeitbedarf im aktuellen Thread simulieren:

```
                Thread.Sleep(200);
                Debug.WriteLine("Rechne + " + i.ToString());
            }
            return res;
        }
...
```

Aufruf:

```
            listBox1.Items.Clear();
            listBox1.Items.Insert(0, "Start");
            int summe = MeineLangeBerechnung();
            listBox1.Items.Insert(0, "Summe = " + summe.ToString());
            listBox1.Items.Insert(0, "Ende");
...
```

Sie sehen schon, die obige Routine müht sich, möglichst langsam zu sein und gibt einen *int*-Wert zurück. Ein Aufruf wird erwartungsgemäß die Oberfläche in den Tiefschlaf versetzen.

Wir wollen es besser machen:

BEISPIEL 9.35: Neuer, asynchroner Code

C#

```
...
        private async Task<int> MeineLangeBerechnungAsync1()
        {
            int res = 0;
            for (int i = 0; i < 50; i++)
            {
                res += i;
```

Der simulierte Zeitbedarf landet jetzt automatisch in einem extra Thread, die Programmoberfläche hat damit genügend Zeit, Nutzereingaben zu verarbeiten:

BEISPIEL 9.35: Neuer, asynchroner Code

```
                await Task.Delay(200);  // Zeitbedarf simulieren
                Debug.WriteLine("Rechne + " + i.ToString());
            }
            return res;
        }
...
```

Die asynchrone Verwendung:

```
            listBox1.Items.Clear();
            listBox1.Items.Insert(0, "Start");
            int summe = await MeineLangeBerechnungAsync1();
            listBox1.Items.Insert(0, "Summe = " + summe.ToString());
            listBox1.Items.Insert(0, "Ende");
...
```

Das war doch gar nicht so schlimm, mit geringsten Änderungen haben Sie asynchron ausführbaren Code produziert, ohne sich mit Threads herumärgern zu müssen.

Eine kleine Änderung können Sie noch an obigem Code vornehmen: In der Schleife der Berechnungsfunktion brauchen Sie jetzt nicht mit *Debug.WriteLine* zu arbeiten, Sie können auch Ausgaben direkt in die aktuelle Oberfläche schreiben, und das ohne einen Threadwechsel:

```
...
                await Task.Delay(200);  // Zeitbedarf simulieren
                listBox1.Items.Insert(0, "Calc" + i.ToString());
                Debug.WriteLine("Rechne + " + i.ToString());
...
```

Doch was, wenn Sie nicht in Ihren alten Libraries herumfrickeln wollen und trotzdem die *async*-Features unterstützen wollen? Auch das ist möglich, in diesem Fall packen wir einfach den ganzen alten Code bzw. den Methodenaufruf in einen extra Thread:

BEISPIEL 9.36: Alten synchronen Code kapseln

C#

```
...
        public Task<int> MeineLangeBerechnungAsync()
        {
            return Task.Factory.StartNew<int>(() => MeineLangeBerechnung());
        }
...
```

Der Aufruf:

```
        int summe = await MeineLangeBerechnungAsync();
```

Damit steht einer Erweiterung Ihrer Libraries um asynchrone Methoden steht nichts mehr im Wege.

9.10 Praxisbeispiele

9.10.1 Spieltrieb & Multithreading erleben

Ein neugieriger Blick auf das Endergebnis dieses Beispiels wird in Ihnen die Vermutung nähren, dass diesmal die Autoren ihrem geradezu lächerlichen Spieltrieb hoffnungslos erlegen sind. Doch wir wollen Ihnen eigentlich nur vermitteln wie Sie

- Objekte mit Threads verbinden,
- Threads initialisieren und parametrieren,
- Threads synchronisieren,
- Thread-Zustände auswerten,
- kritische Abschnitte sichern,
- Callback-Timer ververwenden,
- die Anwendung threadsicher programmieren.

Alles staubtrockene Themen, aber ein bisschen Spaß hilft auch in solch einem Fall die Langeweile zu vertreiben!

HINWEIS: Für den notorischen Besserwisser: Überblättern Sie ganz schnell das "Beispiel", denn Sie haben dafür ganz sicher eine wesentlich effektivere Lösung parat und haben auch bereits erkannt, dass Threads **nicht** die allerbeste Lösung für dieses Problem sind. Uns geht es hier jedoch nur um die Demonstration des Multithreading und nicht um eine superoptimale Realisierung.

Aufgabenstellung

Ausgehend von der Benutzereingabe sollen in einem Lager Kisten hinzugefügt werden. Diese werden von drei LKWs zu einem Schiff transportiert. Ist das Schiff mit fünf Kisten beladen, transportiert es diese weiter, löscht die Ladung und kehrt zurück, um erneut Kisten zu laden usw. Einzige Ausnahme: Ist mindestens eine Kiste an Bord und nach 10 Sekunden keine neue Ladung zu erwarten, legt das Schiff trotzdem ab. Ist das Schiff nicht da, müssen die LKWs natürlich am Hafen warten.

Ganz grob können wir das "Problem" zunächst auf drei Klassen aufteilen:

- LKW
- Schiff
- *Controller* (Lager-Controller)

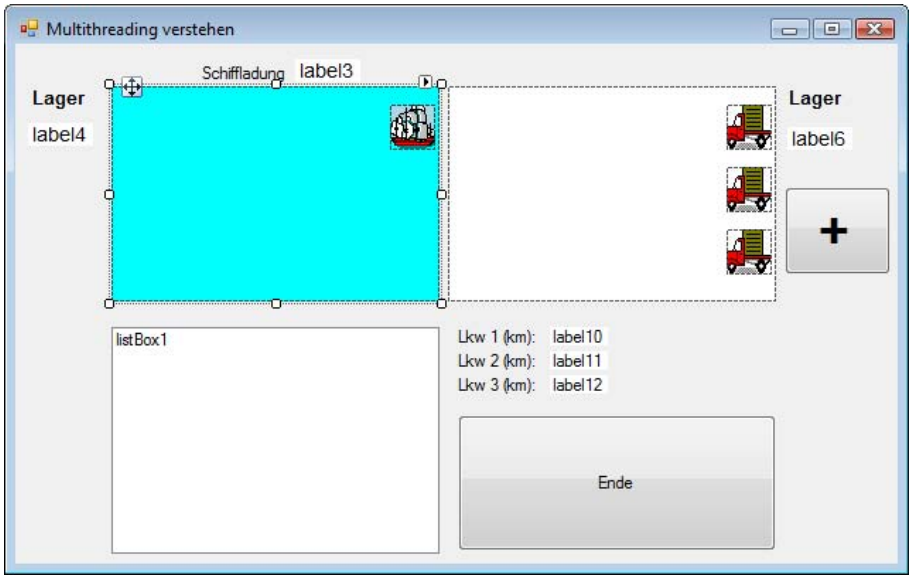
Jede Klasse verfügt über einen fest zugeordneten Thread, auf den wir auch von außen (*public*) zugreifen können.

HINWEIS: Wir werden darauf verzichten, Eigenschaften und enthaltene Objekte zu kapseln, so bleibt der Blick frei für das Wesentliche. Eine threadsichere Programmierung haben wir zwar weitgehend angestrebt, können aber keine Gewähr für die volle Funktionstüchtigkeit unter allen Umständen geben.

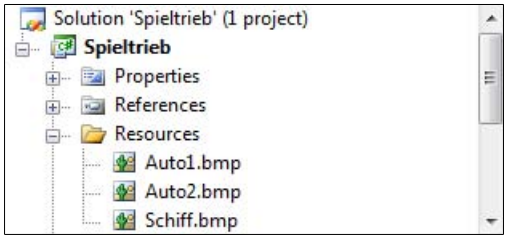
Oberfläche

Auf das Startformular *Form1* einer Windows-Anwendung setzen Sie zwei *Panels*, einige *PictureBox*en sowie diverse *Labels* und *Buttons* entsprechend folgender Abbildung.

HINWEIS: Achten Sie darauf, dass die "LKWs" und das "Schiff" direkt in die Panels eingefügt werden (Clientbereich) und nicht darüber liegen.



Die Bilder selbst fügen Sie als Ressourcen in das Projekt ein:



HINWEIS: Um threadsicher arbeiten zu können, müssen die Ressourcen vor Programmstart in die einzelnen Objekte geladen werden, wir kommen später darauf zurück.

In den *Label*s wird der jeweilige Füllstand bzw. der Kilometerstand angezeigt, die *ListBox* meldet Ihnen wichtige Ereignisse, so behalten Sie immer die Übersicht.

Quelltext LKW

Beginnen wir mit der Klasse *LKW*. Neben seinem *Thread*-Objekt speichert die Klasse noch die zugeordnete Images, d h. die Bildchen für Hin- und Rückfahrt. Die Zuordnung des Bildes aus den Anwendungsressourcen erfolgt bereits mit dem Aufruf des Konstruktors.

Die öffentliche Methode *Abtransport* wird später vom Controller-Objekt ausgelöst, diese startet wiederum den Thread mit der Methode *Fahrt()*.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Drawing;
using System.Drawing.Imaging;

namespace WindowsApplication1
{
    class cLKW
    {
```

Die Fahrzeugnummer:

```
        public int Nr;
```

Die aktuelle Position:

```
        public int Pos;
```

Der Kilometerstand:

```
        public int kmStand;
```

Der Thread:

```
        public Thread myThread;
```

Die Bilder:

```
        private Image Img1;
        private Image Img2;
```

Das aktuelle Bild:

```
        public Image Image;
```

Ein Verweis auf das aufrufende Formular:

```
        private Form1 myForm;
```

Grundzustand initialisieren und *Thread*-Objekt erzeugen:

```
        public cLKW(int nr, Form1 myf)
```

```
        {
            Nr = nr;
            kmStand = 0;
            Pos = 280;
            myForm = myf;
            myThread = new Thread(Fahren);
            Img1 = WindowsApplication1.Properties.Resources.Auto1;
            Img2 = WindowsApplication1.Properties.Resources.Auto2;
            Image = Img1;
        }
```

Die spätere Thread-Methode:

```
        public void Fahren()
        {
```

Bild und Position zuweisen:

```
            Image = Img1;
            Pos    = 280;
```

Die Fahrt beginnt:

```
            for (int posi = 280; posi > 0; posi -= 2)
    {
```

Damit es nicht ganz so schnell geht, werden ein paar Pausen eingelegt:

```
              Thread.Sleep(30);
             Pos = posi;
              kmStand++;
```

Hier rufen wir die zentrale Anzeigeroutine in *Form1* auf:

```
              myForm.FormRefresh("");
      }
```

HINWEIS: Obige Methode greift nicht direkt auf die Steuerelemente auf *Form1* zu, was auch nicht zu empfehlen ist, da der Aufruf aus einem Hintergrund-Thread erfolgen wird. Per *Invoke* erfolgt das Weiterreichen des Aufruf an den Vordergrund-Thread.

Ankunft im Hafen und Warten auf das Schiff:

```
            if (Global.Schiff.myThread.IsAlive) Global.Schiff.myThread.Join();
```

Ist der Thread des Schiffs aktiv[1], können wir davon ausgehen, dass das Schiff unterwegs ist, anderenfalls liegt es im Hafen, und wir können es beladen:

```
            Global.Schiff.Beladen();
```

[1] Um den Thread abfragen zu können, muss das Objekt auch vorhanden sein. Deshalb werden die Threads bereits in den Konstruktoren der Objekte erzeugt.

Rückfahrt (Bildwechsel und langsam nach rechts verschieben):

```
            Image = Img2;
            for (int posi = 0; posi < 280; posi += 2)
            {
                Thread.Sleep(20);
                Pos = posi;
                kmStand++;
                myForm.FormRefresh("");
            }
```

Letzte Statusmeldung für den LKW ausgeben, dann ist der Thread am Ende:

```
            myForm.FormRefresh("Lkw_" + Nr.ToString() + ": Fahrtende");
        }
```

Ruft der Controller die Methode *Abtransport* auf, wird eine Kiste umgeladen (Lagerbestand verringert sich), die Änderung wird angezeigt und der Thread gestartet:

```
        public void Abtransport()
        {
            Global.Controller.Lagerbestand--;
            myForm.FormRefresh("Lkw_" + Nr.ToString() + ": Fahrtbeginn");
            myThread = new Thread(Fahren);
            myThread.Start();
        }
    }
}
```

Quelltext Schiff

Das Schiff muss neben der reinen Fahrt (wie beim LKW) noch eine Zusatzaufgabe erfüllen. Gemeint ist die Abfahrt bei mindestens einer Kiste und 10 Sekunden Inaktivität. Hier bietet sich ein *CallbackTimer* an.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace WindowsApplication1
{
    class Schiff
    {
```

Die Eigenschaften:

```
        public int Ladung;
        public int Transportiert;
        public Form1 myForm;
        public int pos;
```

Timer und *Thread*:

```
public Timer tim;
public Thread myThread;
private int Wartezeit;
```

Der Konstruktor:

```
public Schiff(Form1 mf)
{
```

Position festlegen, *Thread* und *Timer* erzeugen:

```
    pos = 200;
    this.myThread = new Thread(this.Main);
    this.tim = new Timer(OnTimer, null, 0, 3000);
    Wartezeit = 0;
```

Verweis auf *Form1* abspeichern (für die Anzeige der Daten):

```
    myForm = mf;
    Transportiert = 0;
}
```

Die Hauptroutine (Thread-Methode):

```
private void Main()
{
```

Benutzerschnittstelle aktualisieren:

```
    myForm.FormRefresh("Schiff: Abfahrt");
```

Nach links fahren:

```
    for (int posi = 200; posi > 0; posi -=2)
    {
        Thread.Sleep(100);
        pos = posi;
        myForm.FormRefresh("");
    }
```

Im Hafen die Ladung löschen:

```
    myForm.FormRefresh("Schiff: Ladung löschen");
    while (Ladung > 0)
    {
        Thread.Sleep(400);
```

Absichern:

```
        Monitor.Enter(this);
        Ladung--;
        Transportiert++;
        myForm.FormRefresh("");
```

```
            Monitor.Exit(this);
        }
```

Rückfahrt:

```
        for (int posi = 0; posi < 200; posi +=2)
        {
            Thread.Sleep(60);
            pos = posi;
            myForm.FormRefresh("");
        }
        Wartezeit = 0;
        myForm.FormRefresh("Schiff: Warten ...");
    }
```

Starten des Threads:

```
    public void Transport()
    {
        myThread = new Thread(Main);
        myThread.Start();
    }
```

Das Schiff wird durch die LKWs "beladen":

```
    public void Beladen()
    {
        Monitor.Enter(this);
        Wartezeit = 0;
        Ladung++;
        myForm.FormRefresh("Schiff: Beladen (" + Ladung.ToString() + ")");
```

Fünf Kisten an Bord, dann Abtransport:

```
        if (Ladung == 5) Transport();
        Monitor.Exit(this);
    }
```

HINWEIS: Verzichten Sie auf das Sichern des Abschnitts, werden Sie später den visuellen Schwund von Kisten bemerken. Dies ist der Fall, wenn bereits LKWs am Hafen warten (*Join*). Alle LKW-Threads werden gleichzeitig freigegeben und es kommt zu Überschneidungen beim Aufruf der *Beladen*-Methode!

Der *Timer* ist mal wieder abgelaufen:

```
    private void OnTimer(object state)
    {
```

Absichern des Abschnitts, sonst kommen uns noch andere Threads in die Quere:

```
        Monitor.Enter(this);
```

Ist der eigene Thread inaktiv (Hafen!) verlängert sich die Wartezeit:

```
            if (!this.myThread.IsAlive) Wartezeit++;
```

Ist die Wartezeit abgelaufen und mehr als eine Kiste vorhanden:

```
            if ((Wartezeit > 9) & (Ladung > 0))
```

Abfahrt:

```
            {
                myForm.FormRefresh("Schiff: Zeit abgelaufen");
                Wartezeit = 0;
                Transport();
            }
```

Der sichere Abschnitt wird verlassen:

```
            Monitor.Exit(this);
        }
    }
}
```

Quelltext Controller

Hier haben wir es mit "Big Brother" zu tun. Unermüdlich, d h. in einer Endlosschleife, wird der Lagerbestand überprüft und, wenn möglich, auf die LKWs verladen.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace WindowsApplication1
{
    class cController
    {
        public int Lagerbestand;
        public Thread myThread;
```

Wir sind am Leben:

```
        public bool IsAlive;
        private Form1 myForm;
```

Das Objekt startet bereits im Konstruktor seinen Endlos-Thread:

```
        public cController(Form1 mf)
        {
            myForm = mf;
            myThread = new Thread(Main);
```

Wir sind nicht ganz so wichtig, deshalb etwas weniger Rechenzeit:

```
            myThread.Priority = ThreadPriority.BelowNormal;
            myThread.Start();
            IsAlive = true;
        }
```

Unser "Hauptprogramm":

```
        private void Main()
        {
```

Ein paar Aktualisierungen des Formulars:

```
            myForm.FormRefresh("");
            myForm.FormRefresh("Controller: Gestartet");
            int i = 0;
```

Die Endlosschleife:

```
            while (true&IsAlive)  // Bis in alle Ewigkeit
            {
```

Es ist was im Lager:

```
                while (Lagerbestand > 0)
                {
```

Einen der drei LKWs aussuchen[1]:

```
                    if (i > 2) i = 0;
```

Ist der LKW nicht unterwegs (kein laufender Thread), dann beladen:

```
                    if (!Global.LKWs[i].myThread.IsAlive)
                    {
                        myForm.FormRefresh("Controller: Lkw beladen");
                        Thread.Sleep(200);  // Ladezeit
                        Global.LKWs[i].Abtransport();
                    }
                    else
```

Sonst warten wir etwas (0,5 Sekunden) auf diesen LKW:

```
                    {
                        Global.LKWs[i].myThread.Join(500);
```

Ist er jetzt da, dann beladen:

```
                        if (!Global.LKWs[i].myThread.IsAlive)
                        {
                            myForm.FormRefresh("Controller: Lkw beladen");
                            Thread.Sleep(200);  // Ladezeit
                            Global.LKWs[i].Abtransport();
```

[1] Bei den späteren Tests werden Sie feststellen, dass die LKWs durch diese Auswahl recht gleichmäßig belastet werden, kein LKW ist im Dauereinsatz.

```
                    }
                }
                i++;
            }
```

Hier "ruht" sich der Controller etwas aus (Mittagsschlaf):

```
                Thread.Sleep(100);
            }
            myForm.FormRefresh("Controller: Gestoppt");
        }
    }
}
```

Wer aufmerksam war hat sicher erkannt, dass wir es – abgesehen von der Oberfläche – bei obiger Routine mit dem Hauptprogramm zu tun haben. Der Controller wartet auf LKWs und belädt diese. Damit wird der gesamte Kreislauf immer wieder angestoßen sobald Kisten vorhanden sind. Die Oberfläche unserer Anwendung agiert völlig unabhängig von diesen Hintergrundereignissen!

Quelltext Global

Wie Sie sicher bemerkt haben, rufen die Objekte/Klassen untereinander Methoden auf und greifen auf zentrale Objekte zu. Dies lässt sich am einfachsten mit einer statischen Klasse realisieren:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    static class Global
    {
        public static cController Controller;
        public static Schiff Schiff;
        public static cLKW[] LKWs = new cLKW[3];
    }
}
```

Quelltext Form1

Wer hier noch viel Quellcode erwartet, den müssen wir leider enttäuschen. Außer dem Initialisieren der Objekte, dem Erhöhen des Lagerbestands und der Anzeige bleiben keine Aufgaben übrig. Um alles andere kümmern sich die Objekte selbst.

```
namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
```

Für den *Invoke*-Aufruf von *MyRefresh* (auf die Steuerelemente können wir nur aus dem Vordergrund-Thread zugreifen) brauchen wir einen Delegate:

```
private delegate void MyRefreshDelegate(string msg);
private MyRefreshDelegate dg;
```

Die "sichere" Methode für die Anzeige:

```
private void MyRefresh(string msg)
{
```

Nochmal absichern:

```
 lock(this)
 {
```

Handelt es sich um eine Textmeldung, dann nur die *ListBox* aktualisieren:

```
    if (msg != "")
    {
        listBox1.Items.Add(msg);
    }
    else
```

Sonst die allgemeinen Daten abfragen und setzen:

```
    {
        label3.Text = Global.Schiff.Ladung.ToString();
        label4.Text = Global.Schiff.Transportiert.ToString();
        label6.Text = Global.Controller.Lagerbestand.ToString();
        label10.Text = Global.LKWs[0].kmStand.ToString();
        label11.Text = Global.LKWs[1].kmStand.ToString();
        label12.Text = Global.LKWs[2].kmStand.ToString();
        pictureBox2.Image = Global.LKWs[0].Image;
        pictureBox3.Image = Global.LKWs[1].Image;
        pictureBox4.Image = Global.LKWs[2].Image;
        pictureBox1.Left = Global.Schiff.pos;
        pictureBox2.Left = Global.LKWs[0].Pos;
        pictureBox3.Left = Global.LKWs[1].Pos;
        pictureBox4.Left = Global.LKWs[2].Pos;
    }
}
}
```

Für den Aufruf aus den Threads:

```
public void FormRefresh(string msg)
{
```

Fehlerbehandlung ist wichtig, da teilweise noch Meldungen kommen, wenn das Formular schon abgebaut wird (*Close*):

```
    try
    {
```

Und ab mit dem Aufruf in den Vordergrund-Thread:

```
            this.Invoke(dg, new object[] { msg });
        }
        catch
        {
        }
    }
```

Formular und Delegate initialisieren:

```
    public Form1()
    {
        InitializeComponent();
        dg = new MyRefreshDelegate(MyRefresh);
    }
```

Und hier erzeugen wir die Objekte (nicht im Konstruktor, sonst fehlt für den Controller das Fensterhandle):

```
    private void Form1_Load(object sender, EventArgs e)
    {
        Global.LKWs[0] = new cLKW(0, this);
        Global.LKWs[1] = new cLKW(1, this);
        Global.LKWs[2] = new cLKW(2, this);
        Global.Schiff = new Schiff(this);
        Global.Controller = new cController(this);
    }
```

Das ist unser ganzer Eingriff in das System, wir füllen das Lager mit Kisten:

```
    private void button1_Click(object sender, EventArgs e)
    {
        Global.Controller.Lagerbestand++;
    }
```

Und hier kommt unser Sorgenkind. Dass wir exzessiv von Threads Gebrauch machen, rächt sich an dieser Stelle. Der Nutzer drückt irgendwann auf die "Ende"-Schaltfläche. Doch was machen wir mit unseren ganzen "herumlaufenden" Threads?

Ein paar Lösungsansätze zeigt die folgende Routine:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Die LKW-Threads werden "brutal" zerstört:

```
        if (Global.LKWs[0].myThread != null) Global.LKWs[0].myThread.Abort();
        if (Global.LKWs[1].myThread != null) Global.LKWs[1].myThread.Abort();
        if (Global.LKWs[2].myThread != null) Global.LKWs[2].myThread.Abort();
```

Beim Schiff bevorzugen wir die sanfte Variante:

```
        Global.Schiff.tim.Dispose();
        if (Global.Schiff.Ladung > 0)
```

```
            {
                Global.Schiff.Transport();
                MessageBox.Show("Bitte erneut versuchen, Schiff muss noch arbeiten ...");
                return;
            }
            if (Global.Schiff.myThread != null) Global.Schiff.myThread.Abort();
```

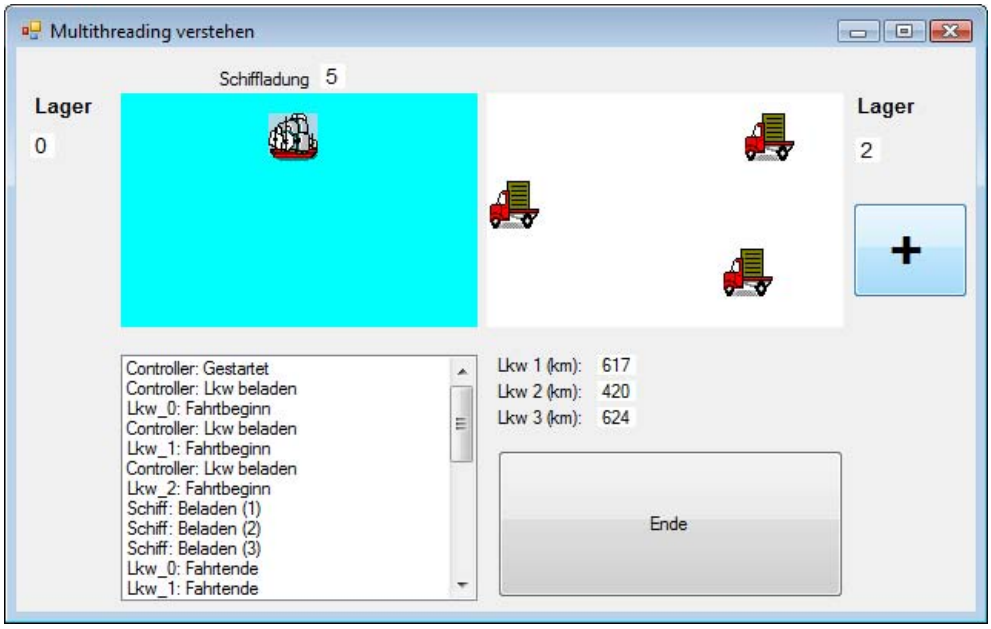
Unseren Controller beenden wir ganz sauber:

```
            Global.Controller.IsAlive = false;
            while (Global.Controller.IsAlive)
                Thread.Sleep(200);
            Close();
        }
    }
}
```

HINWEIS: Es gibt noch eine weitere Methode um die noch existenten Threads auf schnelle Art und Weise loszuwerden. Dazu genügt es, wenn Sie beim Erzeugen des Threads dessen *IsBackground*-Eigenschaft auf *True* setzen. Wird der Vordergrund-Thread (Fenster etc.) beendet, werden automatisch auch alle Hintergrundthreads beendet.

Test

Und jetzt geht es los. Starten Sie das Programm und fügen Sie reichlich Kisten hinzu.



Prüfen Sie, ob auch der Timer funktioniert (nur eine Kiste versenden). Entfernen Sie ruhig einmal die *Monitor*-Objekte und lassen Sie mehrere LKWs am Hafen warten. Sehen Sie sich dabei die "Schiffsladung" an.

Bemerkungen

- Wie bereits eingangs erwähnt: Threads sind **nicht** die ideale Lösung für obiges Problem, da zum einen das Programm schnell an Übersichtlichkeit verliert und zum anderen reichlich Rechenzeit verschwendet wird.
- "Verbesserungen" und Erweiterungen sind natürlich immer möglich. Insbesondere der direkte Zugriff auf die jeweiligen Threads ist nicht empfehlenswert. Auch die Verwendung eines Thread-Pools (z.B. für die LKWs) wäre denkbar.
- Einen wesentlich "moderneren" Lösungsansatz mit den TPL-Sprachkonstrukten von .NET 4.0 bietet das Beispiel 10.10 am Ende des nachfolgenden Kapitels.

9.10.2 Prozess- und Thread-Informationen gewinnen

Dieses Universalbeispiel demonstriert Ihnen nicht nur die Verwendung der *Process*- und *Process-Thread*-Objekte in C#, sondern ist auch für den täglichen Gebrauch des Entwicklers durchaus von Nutzen.

Oberfläche

Wir brauchen zwei *ListView*-Komponenten (*View = Details*), zwei *Label*s und einen *Button* (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Diagnostics;
    ...
```

Unsere Haupt-Referenz auf die *Process*-Klasse (*System.Diagnostics*-Namespace):

```
    private Process ProcObj = new Process();
```

Einige Ergänzungen im Konstruktor-Code:

```
    public Form1()
    {
       ...
       listView1.Columns.Add("Base Name", 110, HorizontalAlignment.Left);
       listView1.Columns.Add("Modules", 70, HorizontalAlignment.Right);
       listView1.Columns.Add("Full Path", 150, HorizontalAlignment.Left);
       listView1.Columns.Add("PID", 40, HorizontalAlignment.Right);
       listView1.Columns.Add("CPU Time", 70, HorizontalAlignment.Right);
       listView2.Columns.Add("State", 100, HorizontalAlignment.Right);
       listView2.Columns.Add("WaitReason", 258, HorizontalAlignment.Left);
       listView2.Columns.Add("CPU Time", 70, HorizontalAlignment.Right);
```

Der Aufruf unserer "Haupt-Methode" *enumProcs()*, die wir im Anschluss implementieren werden:

```
      this.enumProcs();   // nutzerdefinierte Methode
   }
```

Die *enumProc()*-Methode ermittelt alle laufenden Prozesse:

```
   public void enumProcs()
   {
     ListViewItem lvItem;
     System.TimeSpan ts;
```

Laufende Prozesse besorgen und in einem Array ablegen:

```
      Process[] allProcs  = Process.GetProcesses();
```

Alle Prozesse und ihre Beschreibung in *listView1* ausgeben:

```
      for (int i = 0; i < allProcs.Length; i++)
      {
         lvItem = this.listView1.Items.Add(allProcs[i].ProcessName);
```

Alle Prozesse, in denen mindestens ein Modul läuft von den Idle-Prozessen trennen:

```
         try
         {
            lvItem.SubItems.Add(allProcs[i].Modules.Count.ToString());
            lvItem.SubItems.Add(allProcs[i].MainModule.FileName);
         }
         catch
         {
            lvItem.SubItems.Add("0");
            lvItem.SubItems.Add("0");
         }
         finally
         {
            lvItem.SubItems.Add(allProcs[i].Id.ToString());
```

Ein *TimeSpan*-Objekt besorgen:

```
            ts = allProcs[i].TotalProcessorTime;
```

Die Zeit formatieren:

```
            lvItem.SubItems.Add(String.Format("{0:00}", ts.TotalHours) + ":" +
                    String.Format("{0:00}", ts.Minutes) + ":" +
             String.Format("{0:00}", ts.Seconds));
             lvItem.SubItems.Add((allProcs[i].WorkingSet/1000).ToString() + "K");
             label1.Text = "Prozesse: " + allProcs.Length.ToString();
          }
       }
   }
```

Diese Methode ermittelt alle Threads zu einem laufenden Prozess:

```
public void EnumThreads(int ProcID)
{
  System.TimeSpan ts;
  ListViewItem lvItem;
```

Den *ListView*-Inhalt löschen:

```
  listView2.Items.Clear();
```

Den Prozess referenzieren (mittels seiner ID):

```
  Process aProc = Process.GetProcessById(ProcID);
  try
  {
```

Anzeige der Anzahl von Threads:

```
     label2.Text = "Threads: " + aProc.Threads.Count.ToString();
  }
  catch (Exception ex)
  {  MessageBox.Show(ex.Message); }
```

Alle Threads des Prozesses durchlaufen:

```
  foreach (ProcessThread aThread in aProc.Threads)
  {
    lvItem = listView2.Items.Add(aThread.Id.ToString());
    switch (aThread.ThreadState)
    {
       case ThreadState.Initialized:
         lvItem.SubItems.Add("Initialized"); break;
       case ThreadState.Ready:
         lvItem.SubItems.Add("Ready"); break;
       case ThreadState.Running:
         lvItem.SubItems.Add("Running"); break;
       case ThreadState.Standby:
         lvItem.SubItems.Add("Standby"); break;
       case ThreadState.Terminated:
         lvItem.SubItems.Add("Terminated"); break;
       case ThreadState.Transition:
         lvItem.SubItems.Add("In Transition"); break;
       case ThreadState.Unknown:
         lvItem.SubItems.Add("Unkwown"); break;
       case ThreadState.Wait:
         lvItem.SubItems.Add("Waiting"); break;
    }
```

Falls der Thread im Wartezustand ist, soll eine Info ausgegeben werden:

```
    if (aThread.ThreadState == ThreadState.Wait)
      lvItem.SubItems.Add(ListReason(aThread.WaitReason));
```

```
        else
          lvItem.SubItems.Add("N/A");
```

Ein *TimeSpan*-Objekt abholen und die Zeit formatieren:

```
        ts = aThread.TotalProcessorTime;
        lvItem.SubItems.Add(String.Format("{0:00}", ts.TotalHours) + ":" +
            String.Format("{0:00}", ts.Minutes) + ":" + String.Format("{0:00}", ts.Seconds));
      }
    }
```

Die folgende Methode dechiffriert lediglich die *ThreadWaitReason*-Enumeration:

```
  private string ListReason(System.Diagnostics.ThreadWaitReason waitingReason)
  {
    string s = "";
    switch (waitingReason)
    {
      case ThreadWaitReason.EventPairHigh:
        s = "Waiting For Event Pair High"; break;
      case ThreadWaitReason.EventPairLow:
        s = "Waiting For Event Pair Low"; break;
      case ThreadWaitReason.ExecutionDelay:
        s = "Execution Delay"; break;
      case ThreadWaitReason.Executive:
        s = "Waiting for Scheduler"; break;
      case ThreadWaitReason.FreePage:
        s = "Waiting For Free Virtual Mem. Page"; break;
      case ThreadWaitReason.LpcReceive:
        s = "Waiting For A Local Proc. Call To Arrive"; break;
      case ThreadWaitReason.LpcReply:
        s = "Waiting For A Reply To A Local Proc. Call"; break;
      case ThreadWaitReason.PageIn:
        s = "Waiting For Virtual Mem. Page To Arrive In Memory"; break;
      case ThreadWaitReason.PageOut:
        s = "Waiting For Virtual Mem. Page To Write To Disk"; break;
      case ThreadWaitReason.Suspended:
        s = "Execution Suspended"; break;
      case ThreadWaitReason.SystemAllocation:
        s = "Waiting For A System Allocation"; break;
...
      default: break;
    }
    return s;
  }
```

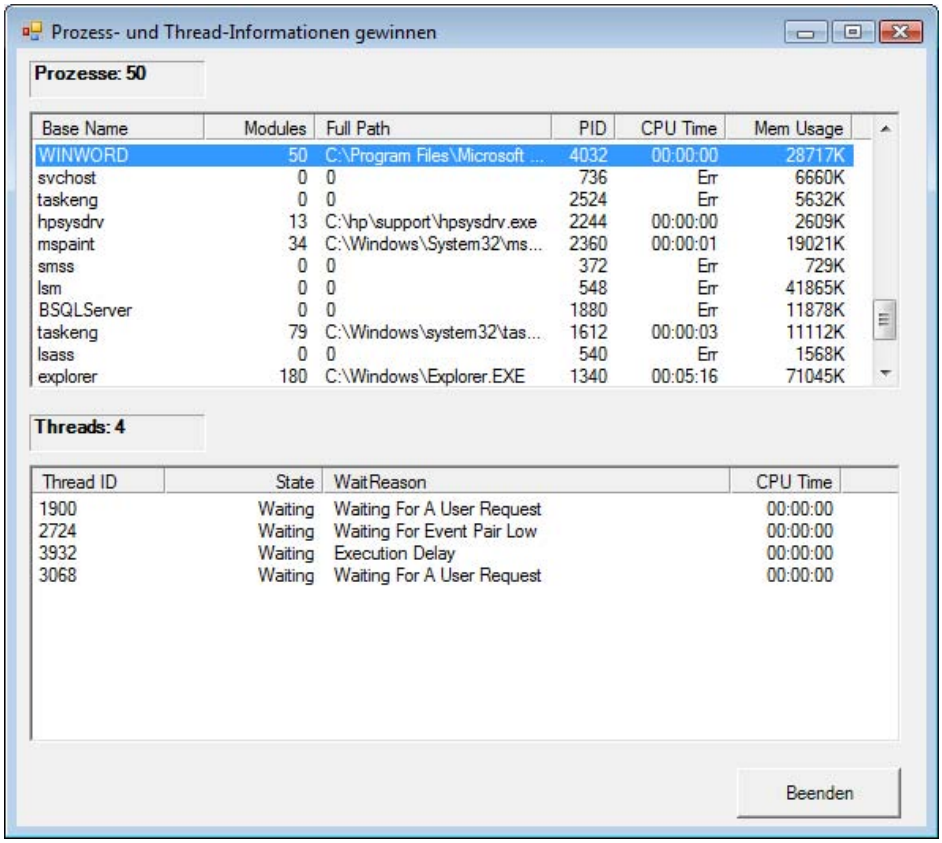
Ein neuer Prozess wird angeklickt:

```
  private void listView1_SelectedIndexChanged(object sender, System.EventArgs e)
  {
    try
```

```
      { EnumThreads(Convert.ToInt32(listView1.SelectedItems[0].SubItems[3].Text)); }
    catch
      {}
    }
  ...
}
```

Test

Nach Programmstart werden alle laufenden Prozesse oben aufgelistet. Klicken Sie auf einen Prozess, so werden die zugehörigen Threads angezeigt.



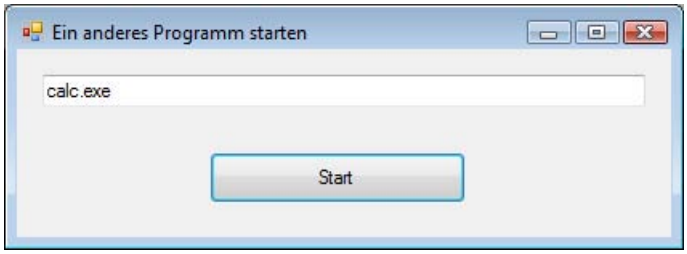
In der Abbildung erkennen Sie, dass momentan 50 Prozesse auf dem PC laufen und das Programm Word (mit dem der Autor gerade **nicht** arbeitet) hier 50 Module beansprucht und in 4 verschiedenen Threads läuft.

9.10.3 Ein externes Programm starten

Möchten Sie aus Ihrer .NET-Anwendung heraus andere Prozesse[1], d h. Programme, starten, bieten sich Ihnen unter C# zahlreiche Möglichkeiten und Optionen an. Das vorliegende Beispiel soll Ihnen einen schnellen Einstieg vermitteln.

Oberfläche

Erstellen Sie eine Oberfläche entsprechend folgender Abbildung:



Über die *TextBox* kann zur Laufzeit Name und Pfad für eine Anwendung eingegeben werden.

Quelltext

Mit Klick auf den *Button* soll folgender Code ausgeführt werden:

```
public class Form1 : System.Windows.Forms.Form
{
    ...
    private void Button1_Click(object sender, System.EventArgs e)
    {
```

Erzeugen eines *Process*-Objekts:

```
        Process proc = new Process();
```

Parametrieren (welche Anwendung soll gestartet werden) und Aufruf:

```
        proc.StartInfo.FileName = textBox1.Text;
        proc.Start();
```

HINWEIS: Diese Methode wartet nicht auf das Prozess-Ende, die Programmausführung wird direkt fortgesetzt.

Mit der folgenden Methode erreichen wir, dass die Anwendung solange wartet, bis der Prozess beendet ist:

```
         proc.WaitForExit();
```

[1] Hier ist wirklich von Prozessen und nicht Threads die Rede.

```
            MessageBox.Show("Fertig");
        }
}
```

HINWEIS: Möchten Sie die Anwendung bzw. den Prozess asynchron starten, lassen Sie einfach die Methode *WaitForExit* weg.

Test

Geben Sie in die *TextBox* einen Anwendungsnamen (z.B. *Calc.exe*, *MSPaint.exe*, ...) ein und klicken Sie auf die "Start"-Schaltfläche (siehe folgende Abbildung).

HINWEIS: Die Messagebox mit der Meldung "Fertig" sollte erst angezeigt werden, wenn die aufgerufene Anwendung beendet ist.

Ergänzungen

Eine besonders einfache Variante bietet sich mit dem direkten Aufruf der Methode *Start* an:

```
System.Diagnostics.Process.Start("calc.exe");
```

Beachten Sie jedoch, dass Sie in diesem Fall nicht mit *WaitForExit* auf das Programmende warten können, für viele Anwendungszwecke reicht diese Verfahrensweise jedoch aus

Über die *StartInfo*-Eigenschaft können Sie unter anderem folgende Optionen für den zu startenden Prozess vorgeben:

Eigenschaft	Beschreibung
Arguments	... die Kommandozeilenparameter für den neuen Prozess (beispielsweise eine zu öffnende Datei)
CreateNoWindow	... Start in einem neuen Fenster (*true/false*)
FileName	... die eigentliche Anwendung
WindowStyle	... der Startmodus für das Anwendungsfenster (maximiert, minimiert, versteckt etc.)
WorkingDirectory	... das Arbeitsverzeichnis der Anwendung

BEISPIEL 9.37: Verwendung von Argumenten beim Aufruf der *Start*-Methode

C#
```
System.Diagnostics.Process.Start("notepad.exe",@"c:\test.txt");
```

BEISPIEL 9.38: Eine Webseite im Explorer öffnen

C#
```
System.Diagnostics.Process.Start("http:/www.microsoft.com");
```

BEISPIEL 9.39: Eine E-Mail erzeugen

C#
```
System.Diagnostics.Process.Start("mailto:max_musterman@nirgendwo.de");
```

Bemerkung

In diesem Beispiel haben wir Ihnen die grundsätzliche Methodik zum Starten von externen Anwendungen vorgestellt, C# bietet jedoch auch die Möglichkeit, das Ende des Prozesses mit einem Ereignis zu überwachen. Der Vorteil: Sie können mit Ihrer Anwendung normal weiterarbeiten (beim synchronen Ausführen wird nicht einmal das Fenster aktualisiert) und dennoch auf das Prozess-Ende reagieren[1].

[1] Weitergehende Informationen finden Sie in unserem [Visual C# 2012 Kochbuch].

Kapitel 10

Die Task Parallel Library

Nachdem wir Sie im vorhergehenden Kapitel bereits intensiv mit den Möglichkeiten der asynchronen Programmierung und deren Grundlagen vertraut gemacht haben, wollen wir uns nun ausgiebig mit den praktischen Möglichkeiten der *Task Parallel Library* beschäftigen.

10.1 Überblick

Sicher stellt sich Ihnen jetzt die Frage, warum Sie sich überhaupt mit den Multithreading-Sprachkonstrukten herumplagen sollen, wenn es davon doch bereits ein reichliches Angebot gibt.

Der Vorteil den Ihnen die *Task Parallel Library* liefert: Sie müssen sich als Programmierer keinen Kopf über die spätere Anzahl von Prozessoren machen, die Library passt die Anzahl der erzeugten Threads automatisch an die Anzahl der verfügbaren Prozessorkerne an. Ihre Anwendung ist also z.B. nicht explizit für ein 4-Kern-System kompiliert, sondern funktioniert auch problemlos auf einem Einkern-Prozessor (wenn auch langsamer). Sinnloser Verwaltungsoverhead durch zu viele Threads fällt in diesem Fall jedoch nicht an. Würden Sie die gleiche Aufgabenstellung mit den Mitteln von .NET 2.x bzw. 3.x realisieren, müssten Sie sich selbst darum kümmern, die Threadanzahl an die Anzahl der verfügbaren Prozessorkerne anzupassen und die Prioritäten der einzelnen Threads zu bestimmen. All diese Aufgaben werden intern über den CLR-Threadpool abgewickelt, wir kommen auf dieses Thema später noch einmal zurück.

10.1.1 Parallel-Programmierung

Sicher ist Ihnen beim Blick in diverse Computer-Prospekte bereits aufgefallen, dass sich in den neueren Systemen mittlerweile meist mehr als ein Prozessor bzw. Prozessorkern befindet. Systeme mit zwei oder vier Kernen (teilweise zusätzlich mit Hyperthreading) sind keine Exoten mehr, was die Frage aufwirft, wie Ihre Programme von diesem Mehr an Prozessorleistung profitieren können.

Ganz allgemein kann hier auf das bereits angesprochene Multithreading verwiesen werden, mehrere Threads in Ihrem Programm werden im Zweifel auch auf mehrere Prozessorkerne verteilt. Allerdings hat die explizite Verwendung von Threads auch Nachteile. So müssen Sie zum Beispiel wissen, wie viele Prozessorkerne vorhanden sind, um Ihr Programm optimal an diese Anzahl anzu-

passen. Der Aufwand, eine komplexe Aufgabe in mehrere Threads zu zerlegen, kann schnell ausufern und das erzeugte Programm ist meist nur noch für den Profi durchschaubar.

An dieser Stelle kommt das Konzept der "Parallel-Programmierung" ins Spiel: Ohne dass Sie sich um das Erstellen und Verwalten der Threads kümmern müssten, lassen sich wiederkehrende und voneinander unabhängige Aufgaben parallel abarbeiten und damit wesentlich beschleunigen. Als Beispiel sei an dieser Stelle die Verarbeitung von Arrays und Collections genannt, die entsprechenden Sprachkonstrukte *Parallel.For* und *Parallel.ForEach* besprechen wir ab Seite 509 im Detail.

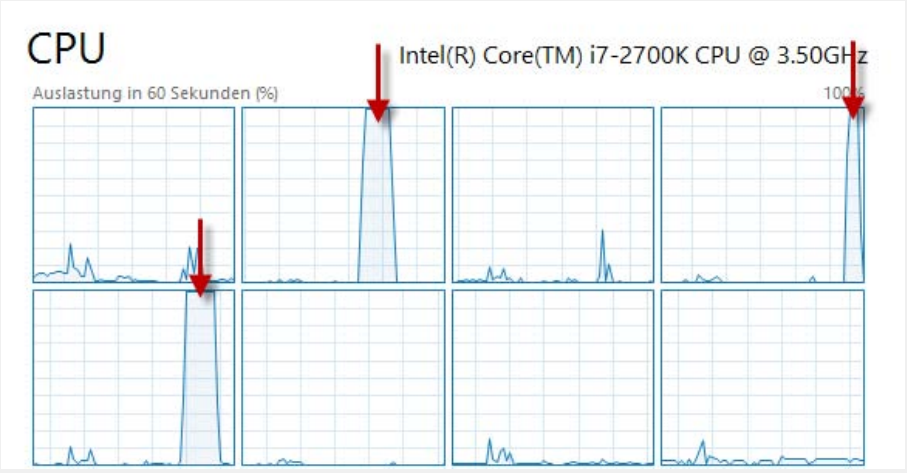
BEISPIEL 10.1: Unterschiede bei der Verarbeitung einer Schleife

C#

Die konventionelle Version mit einer ganz normalen *for*-Schleife:

```
double res = 0;
for (int i = 0; i < 10000000; i++)
   res += (Math.Sin(new Random().NextDouble()) * Math.Acos(new Random().NextDouble()));
```

Die Laufzeit beträgt auf einem 3,5 GHz i7 ganze 22.897 ms bei recht ungleichmäßiger Prozessorlast. Die Aufgabe wechselt im Laufe der Zeit den Kern des Prozessors, die restlichen Kerne werden in dieser Zeit nicht für die Berechnung genutzt:



Beachten Sie, dass obige Prozessornutzung zeitversetzt ist, d h. nur immer ein Prozessor läuft auf "Volllast".

Die parallelisierte Version (zu den Details später mehr):

```
double res = 0;
Parallel.For(0, 4000000, i =>
  {
    res += (Math.Sin(new Random().NextDouble()) * Math.Acos(new Random().NextDouble()));
  });
```

BEISPIEL 10.1: Unterschiede bei der Verarbeitung einer Schleife

Die Laufzeit beträgt jetzt nur noch 6.128 ms (rund ein Viertel) bei nahezu 100% Prozessorlast und gleichmäßiger Verteilung auf alle Kerne:

Je mehr Kerne bzw. Prozessoren, desto schneller kann jetzt die Aufgabe realisiert werden.

HINWEIS: Um Missverständnissen gleich vorzubeugen: Eine Anwendung wird durch die Verwendung der doppelten Anzahl von Prozessoren nicht automatisch doppelt so schnell. Zum einen profitiert Ihre Anwendung nur im parallelisierten Abschnitt davon, zum anderen ist das im Hintergrund erforderliche Erzeugen und Verwalten von Threads auch mit zeitlichem Aufwand verbunden[1]. Ganz nebenbei muss die Anwendung in den meisten Fällen auch mit dem Nutzer kommunizieren.

Leider wird die auf den ersten Blick einfache Verwendung dieser parallelen Sprachkonstrukte auch zu einem "Parallelisierungswahn" führen, bei dem viele Entwickler vergessen, dass bei der parallelen Verarbeitung von Aufgaben auch gewissen Spielregeln eingehalten werden müssen. So kann das Parallelisieren von Zugriffen auf mehrere Webserver bei entsprechenden Antwortzeiten recht sinnvoll sein, der parallele Zugriff auf die Festplatte des System jedoch zu unnötigem "Festplattenrasseln" (Positionierungen des Schreib-Lesekopfes) führen, im Extremfall wird Ihre Anwendung dadurch nicht schneller sondern eher langsamer.

Ganz anders kann dies jedoch beim Parallelisieren einer komplexen Berechnung aussehen. Sollen beispielsweise die Koordinaten eines 3D-Modells neu berechnet werden, bietet sich diese Aufgabe fast schon automatisch für eine Parallelisierung an: Jede Koordinate kann unabhängig von den anderen berechnet werden. Hier führt schon der Ersatz des bisherigen *For* durch ein *Parallel.For* zu einer wesentlichen Beschleunigung der **Berechnung**. Beachten Sie jedoch auch in diesem Fall,

[1] Mehr dazu unter *http //de.wikipedia.org/wiki/Amdahlsches_Gesetz*

dass meist auch noch eine Anzeige erforderlich ist, und da verliert sich der Zeitvorteil teilweise wieder.

Auf den Punkt "unabhängige Aufgaben" sollten wir auch noch kurz eingehen. Lässt sich eine Aufgabe nicht in voneinander unabhängige Aufgaben zerlegen (siehe obiges Beispiel), vergessen Sie die Parallelisierung ganz schnell wieder. Das in diesem Fall erforderliche Sperren von Objekten und deren Freigabe kann ganz schnell im Chaos (Deadlocks, Racing) enden, von längeren Laufzeiten ganz zu schweigen.

10.1.2 Möglichkeiten der TPL

Ausgehend von den beiden Namenspaces

- *System.Threading* und
- *System.Threading.Tasks*

bietet Ihnen die TPL[1] folgende neue bzw. erweiterte Möglichkeiten:

- Parallele Verarbeitung von Methoden und deren Synchronisation mit *Parallel.Invoke*
- Parallele Verarbeitung von Arrays und Collections mit *Parallel.For* und *Parallel.ForEach*
- Einführung der *Task*-Klasse als Ersatz für *Thread* bei automatischer Verwendung des Threadpools
- Die Klassen *Barrier, CountdownEvent, ManualResetEventSlim, SemaphoreSlim, SpinLock* und *SpinWait* für die Synchronisation von Threads
- Parallel-LINQ als parallel verarbeitendes Pendant zu LINQ to Objects

In den folgenden Abschnitten wollen wir uns mit den Details und der Verwendung der TPL beschäftigen, doch bevor es soweit ist, werfen wir noch schnell einen Blick auf den CLR[2]-Threadpool.

10.1.3 Der CLR-Threadpool

Die grundsätzliche Aufgabe des internen CLR-Threadpools ist die möglichst effiziente Verteilung der angeforderten Aufgaben (Tasks/Threads) auf die vorhandene Hardware, d.h. auf die verfügbaren Prozessoren bzw. Cores. Im Vordergrund steht die Vermeidung von unnötigem Overhead beim Erzeugen bzw. Zerstören der Threads. Beide Aufgaben erfordern im Normalfall einen relativ hohen Verwaltungsaufwand und entsprechende Prozessorzeit.

Ganz nebenbei sorgt der CLR-Threadpool auch dafür, dass mit einem recht effizienten Algorithmus anstehende Aufgaben so verteilt werden, dass kein Prozessor bzw. Kern im "Leerlauf" ist, d h., sind die für einen Kern anstehenden Aufgaben abgearbeitet und hat ein anderer Kern noch ausstehende

[1] Sie ahnen es sicher, es handelt sich um die *Task Parallel Library*.

[2] *Common Language Runtime* = NET-Laufzeitumgebung

Aufgaben, werden diese an den freien Kern übergeben. In der einschlägigen Literatur finden Sie dafür den Begriff "work stealing".

Anders als beim bisherigen Einsatz des Threadpools wird bei Verwendung der TPL ein zusätzlicher lokaler Thread-Pool erzeugt, dessen anstehende Aufgaben in LIFO-Art (letzte Aufgaben zuerst) abgearbeitet werden. Der Hintergrund für dieses Verhalten ist eine optimalere Verwendung der bereits im Cache befindlichen Daten.

HINWEIS: Diese Verfahrensweise können Sie über das *PreferFairness*-Flag des übergeordneten Tasks beeinflussen.

Auf weitere interne Details des Threadpools wollen wir nicht weiter eingehen, ist es doch die Aufgabe der TPL, den Programmierer möglichst nicht mit derartigen Feinheiten zu belästigen.

10.2 Parallele Verarbeitung mit Parallel.Invoke

Wer sich durch das vorherige Kapitel gearbeitet hat, ist bereits mit vielen Varianten vertraut, wie eine oder mehrere Methoden in einem extra Thread ausgeführt werden können. Die vorliegende TPL-Lösung dürfte auf Grund ihrer einfachen Verwendbarkeit aber alle anderen Alternativen in den Schatten stellen. Doch gleich ein wichtiger Hinweis vorweg:

HINWEIS: *Parallel.Invoke* kann nicht dazu verwendet werden, einen zum Hauptthread parallelen Thread zu erzeugen. *Parallel.Invoke* blockiert den Ablauf solange, bis **alle** übergebenen Methoden abgearbeitet sind (synchonisiert) und setzt erst dann den Programmablauf fort.

Damit dürfte sich auch gleich das klassische Szenario für die Verwendung der Methode ableiten: Sie haben eine Reihe von Aufgaben, die teilweise zeitintensiv sind und die sich jedoch nicht gegenseitig beeinflussen. Erst wenn alle Aufgaben abgearbeitet sind, wird die Programmausführung fortgesetzt (Datenbankinitialisierung, Abfrage von Webdiensten, Laden von Grafiken, Durchsuchen des Dateisystems etc.).

Die Syntax:

SYNTAX:
```
Parallel.Invoke(array<Action>[])
```

Übergeben Sie einfach eine Reihe von Methodennamen bzw. nutzen Sie die Möglichkeit, gleich Lambda-Ausdrücke aufzurufen. Doch nun zur praktischen Verwendung.

10.2.1 Aufrufvarianten

Einige Beispiele zeigen Ihnen mögliche Aufrufvarianten:

BEISPIEL 10.2: Parallele Verarbeitung von drei Methoden

C#

Namespace einbinden:

```
using System.Threading;
using System.Threading.Tasks;
...
```

Die drei Methoden definieren (ohne sinnvolle Aufgabe):

```
private void AufgabeA()
{ Thread.Sleep(5000); }

private void AufgabeB()
{ Thread.Sleep(5000); }

private void AufgabeC()
{ Thread.Sleep(5000); }
```

Der eigentliche Aufruf:

```
private void Form1_Load(object sender, EventArgs e)
{
    MessageBox.Show("Methoden parallel aufrufen ...");
    Parallel.Invoke(AufgabeA, AufgabeB, AufgabeC);
    MessageBox.Show("Alle Methoden abgearbeitet!");
}
```

Sie werden feststellen, dass die Verarbeitungsdauer ungefähr fünf Sekunden beträgt, bei sequenzieller Abarbeitung sind es erwartungsgemäß 15 Sekunden.

Die oben gezeigte Lösung kann problemlos auch 5 oder 10 Methoden aufrufen, alternativ lassen sich die zu verarbeitenden Methoden auch per Array übergeben, so haben Sie beispielsweise zur Laufzeit die Möglichkeit, die gewünschten Methoden festzulegen.

BEISPIEL 10.3: Alternative zum vorhergehenden Beispiel, jedoch Übergabe per *System.Action*-Array

C#

```
...
    System.Action[] aufgaben = new System.Action[] { AufgabeA, AufgabeB, AufgabeC };
    Parallel.Invoke(aufgaben);
...
```

Selbstverständlich können Sie auch direkt Lambda-Ausdrücke an die *Parallel.Invoke*-Methode übergeben, wie es das folgende Beispiel zeigt:

BEISPIEL 10.4: Parallele Verarbeitung dreier Lambda-Ausdrücke

C#
```
...
            Parallel.Invoke(() => { Thread.Sleep(3000); },
                            () => { Thread.Sleep(5000); },
                            () => { Thread.Sleep(2000); });
...
```

Dass aus den Lambda-Ausdrücken heraus auch wieder Methoden aufgerufen werden können, dürfte sicher bekannt sein.

10.2.2 Einschränkungen

Folgende Einschränkungen sollten Sie bei Verwendung von *Parallel.Invoke* beachten:

- Es muss sich um parameterlose Methoden/Delegates handeln, die keinen Rückgabewert haben.
- Die Methoden dürfen nicht gleichzeitig auf die gleichen Variablen zugreifen.
- Beim Zugriff auf gemeinsame Ressourcen müssen Sie Deadlocks vermeiden.
- Aus den aufgerufenen Methoden dürfen Sie nicht direkt auf das User-Interface zugreifen (dieses läuft in einem anderen Thread.

HINWEIS: Wie Sie sehen, handelt es sich wieder um die gleichen allgemeinen Belehrungen, wie sie auch schon im vorhergehenden Kapitel eine Rolle gespielt haben.

Damit sind wir bereits beim nächsten Sprachkonstrukt angelangt.

10.3 Verwendung von Parallel.For

Und hier haben wir es auch schon mit einem sehr interessantesten Konstrukt zu tun, das neben *Parallel.ForEach* eine konkurrenzlos einfache Möglichkeit bietet, aufwändige Berechnungen zu parallelisieren und damit auf geeigneten Systemen teilweise dramatisch zu beschleunigen, ohne sich große Gedanken um Multithreading etc. machen zu müssen.

Der grundsätzliche Aufruf entspricht in etwa einer *for*-Anweisung (Ähnlichkeiten sind nicht zufällig, sondern beabsichtigt):

SYNTAX: `Parallel.For(startIndex, endIndex, (currentIndex) => { ... Body ... });`

Ein kleines Beispiel soll Ihnen das Grundprinzip demonstrieren.

BEISPIEL 10.5: Parallele Verarbeitung eines Arrays

C#
```
...
using System.Threading;
using System.Threading.Tasks;
```

BEISPIEL 10.5: Parallele Verarbeitung eines Arrays

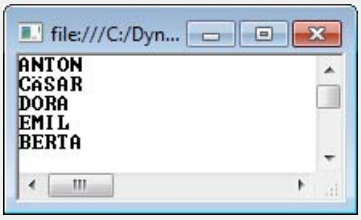
```
...
        static void Main(string[] args)
        {
            string[] Vornamen = { "anton", "berta", "cäsar", "dora", "emil", "friedrich" };
            Parallel.For(0, 5, (i) =>
               {
                   Console.WriteLine(Vornamen[i].ToUpper());
               });
            Console.ReadKey();
        }
```

Wie Sie sehen, steht dem Lambda-Ausdruck ein Parameter *i* zur Verfügung, über den der betreffende Index identifiziert werden kann.

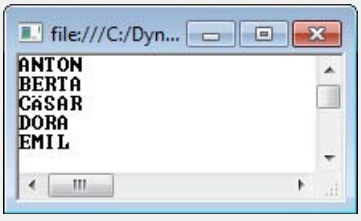
Der Code des obigen Lambda-Abschnitts wird gleichmäßig auf alle Kerne aufgeteilt, je mehr es davon gibt, umso besser.

Ergebnis

Im Folgenden sehen Sie das Ergebnis des ersten Programmaufrufs:



Ein weiterer Aufruf zeigt jedoch folgendes Ergebnis:



Bei genauem Vergleich werden Sie feststellen, dass die Reihenfolge der Ausgabe nicht mit der Reihenfolge im Array übereinstimmt. Die Ausgabe ist vielmehr rein zufällig, da die einzelnen Aufgaben vom internen Threadpool frei an die verfügbaren Prozessorkerne verteilt werden.

BEISPIEL 10.6: Nur zum Vergleich die konventionelle Lösung per for-Schleife:

C#

```
...
            for (int i = 0; i < 6; i++)
            {
```

BEISPIEL 10.6: Nur zum Vergleich die konventionelle Lösung per for-Schleife:

```
                Console.WriteLine(Vornamen[i].ToUpper());
            }
...
```

In diesem Fall wird lediglich ein Prozessor-Kern ausgelastet, die restlichen Kerne stehen für die obige Aufgabe nicht zur Verfügung, was im Zweifel zu einer deutlich längeren Laufzeit führt[1].

10.3.1 Abbrechen der Verarbeitung

Wie bei jeder *for*-Schleife besteht in einigen Situationen manchmal der Wunsch nach einem vorzeitigen Abbruch der Schleife. Mit einer solchen haben wir es in diesem Fall aber gar nicht zu tun, sondern vielmehr mit einer internen Liste von Threads, die durch den internen Threadpool abgearbeitet werden.

Um Einfluss auf die Verarbeitung nehmen zu können, bietet *Parallel.For* eine weitere Überladung, die einen zusätzlichen Parameter an den Lambda-Ausdruck übergibt:

SYNTAX: `Parallel.For(startIndex, endIndex, (currentIndex, loopState) => { ... Body ... });`

BEISPIEL 10.7: "Schleifenabbruch" realisieren

C#

```
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
            string[] Vornamen = { "anton", "berta", "cäsar", "dora", "emil", "friedrich" };
            Parallel.For(0, 5, (i, loopstate) =>
               {
```

Vorzeitiges Ende, wenn in einer Zelle der Name "dora" gefunden wird:

```
                if (Vornamen[i] == "dora") loopstate.Stop();
                if (loopstate.IsStopped) Console.Write("STOP ");
                Console.WriteLine(Vornamen[i].ToUpper());
            });
...
```

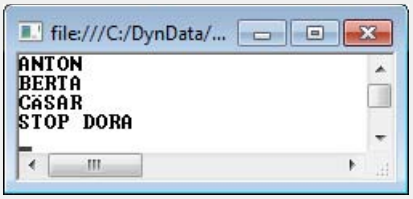
Es ist nicht sicher, dass nach der Verarbeitung des aktuellen Threads nicht doch noch ein anderer Thread ausgeführt wird, wie es auch die folgenden Abbildungen zeigen.

[1] Das muss nicht unbedingt auf unser "anspruchsvolles" Beispiel zutreffen, etwas mehr Logik sollte schon in den betreffenden Abschnitten parallelisiert werden.

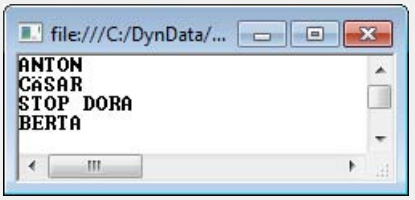
BEISPIEL 10.7: "Schleifenabbruch" realisieren

Ergebnis

Die optimale Version:



Ein Problemfall:



HINWEIS: Über den *ParallelLoopState*-Parameter können Sie auch Informationen über Fehler in den parallelen Threads abrufen. Verwenden Sie dazu die Eigenschaft *ParallelLoopState.IsExceptional*.

10.3.2 Auswerten des Verarbeitungsstatus

Sicher ist es für den Programmierer in einigen Fällen interessant, den Verarbeitungsstatus nach dem Durchlaufen der "Schleife" zu bestimmen. Hier hilft Ihnen die *ParallelLoopResult*-Struktur weiter, die von *Parallel.For* zurückgegeben wird.

BEISPIEL 10.8: Verwendung von *ParallelLoopResult*

C#

```
var res = Parallel.For(0, 5, (i, loopstate) =>
   {
       if (Vornamen[i] == "dora") loopstate.Break();
       if (loopstate.IsStopped) Console.Write("BREAK ");
       Console.WriteLine(Vornamen[i].ToUpper());
   });
```

Wurden alle Durchläufe erfolgreich abgeschlossen, können Sie dies mit der Eigenschaft *IsCompleted* bestimmen:

```
if (res.IsCompleted)
    Console.WriteLine("Kein Abbruch");
else
```

Falls nicht, können Sie auswerten wann der Abbruch erfolgte:

```
   Console.WriteLine("Abbruch bei " + res.LowestBreakIteration.Value.ToString());
Console.ReadKey();
```

HINWEIS: Wird über *IsCompleted false* zurückgegeben und gibt *LowestBreakIteration null* zurück, wurde die *Stop*-Methode aufgerufen. Bei einem Aufruf von *Break* können Sie über *LowestBreakIteration* den betreffenden Indexwert auslesen.

HINWEIS: Es gibt noch weitere Überladungen der *Parallel.For*-Methode, auf die wir jedoch im Rahmen dieses Buchs nicht eingehen können.

10.3.3 Und was ist mit anderen Iterator-Schrittweiten?

Sicher wird der eine oder andere beim Konvertieren seines bisherigen Schleifencodes auch über den Fakt stolpern, dass bei *Parallel.For* nur Long/Integer-Werte für den Iterator zulässig sind. Mit einer kleinen zusätzlichen Berechnung können Sie sich aber schnell weiterhelfen[1]:

BEISPIEL 10.9: Iteration mit einer Schrittweite von 0,001 zwischen 0 und 1 realisieren

C#
```
Parallel.For(0, 1000, i =>
  {
      double d = i / 1000.0;
      MeineMethode(d);
  });
```

HINWEIS: Ganz nebenbei sei auch erwähnt, dass Sie nur positive Iterationen realisieren können, d h., der Startindex muss kleiner als der Endindex sein.

10.4 Collections mit Parallel.ForEach verarbeiten

Neben der indexbasierten Verarbeitung mit *Parallel.For* bietet sich mit *Parallel.ForNext* der Pendant für den parallelen Zugriff auf Collections an.

SYNTAX:
```
Parallel.ForEach<(Of <(TSource)>)>(IEnumerable<(Of <(TSource)>)>),
                                   (Element) => { ... Body ... }))
```

Übergabewerte sind in diesem Fall eine Collection vom Typ *IEnumerable* sowie ein Lambda-Ausdruck mit dem einzelnen Element als Parameter.

BEISPIEL 10.10: Unser Trivialbeispiel aus dem vorhergehenden Kapitel mittels *Parallel.ForEach* realisiert

C#
```
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
            string[] Vornamen = { "anton", "berta", "cäsar", "dora", "emil", "friedrich" };
```

[1] Beachten Sie jedoch, dass in unserem Fall etwas Prozessorzeit für die Division "verschwendet" wird, dies aber sollte den Vorteil der parallelen Verarbeitung nicht aufheben.

BEISPIEL 10.10: Unser Trivialbeispiel aus dem vorhergehenden Kapitel mittels *Parallel.ForEach* realisiert

Der Aufruf gestaltet sich einfacher als bei der *Parallel.For*-Lösung:

```
            Parallel.ForEach(Vornamen, vn =>  {
                Console.WriteLine(vn.ToUpper());
            });
            Console.ReadKey();
...
```

HINWEIS: Weitere Ausführungen zu *Parallel.ForEach* sparen wir uns, da wir an dieser Stelle auf den Abschnitt 10.3 verweisen können. Die Möglichkeiten zum Abbruch und zur Auswertung entsprechen den dort vorgestellten Lösungen.

10.5 Die Task-Klasse

Haben Sie das vorhergehenden Kapitel eingehend studiert, dürfte Ihnen die Verwendung bzw. das Konzept der *Task*-Klasse[2] nicht unbekannt vorkommen. Diese bietet in etwa die gleichen Möglichkeiten wie die schon bekannte *Thread*-Klasse, das allerdings auf einem höheren Abstraktionsniveau und, im Gegensatz zum Thread, unter Verwendung des CLR-Threadpools. Neben dem einfacheren Handling ermöglicht diese Klasse auch eine bessere Fehlerbehandlung.

10.5.1 Einen Task erzeugen

Eine Instanz der *Task*-Klasse können Sie entweder über den gewohnten Weg mittels *new* oder über die statische *Task.Factory.StartNew*-Methode erzeugen. Während Sie bei *new* den Task explizit starten müssen/können, wird in letzterem Fall der Task bereits automatisch gestartet.

Für die Parametrierung bzw. die Zuweisung des Codes per Delegate bieten sich drei Varianten an:

- Zuweisen eines benannten Delegates,
- Übergabe einer anonymen Methode,
- Verwendung eines Lambda-Ausdrucks, dieser kann wiederum benannte Methoden aufrufen.

BEISPIEL 10.11: Die drei Möglichkeiten für die Zuweisung des Task-Codes

C#

```
using System.Threading;
using System.Threading.Tasks;
...
```

Direkter Methodenaufruf:

```
        static void test()
        {
```

[2] Hier ist mit "Task" die Aufgabe und nicht der Task aus dem Task-Manager gemeint.

BEISPIEL 10.11: Die drei Möglichkeiten für die Zuweisung des Task-Codes

```
            Console.WriteLine("Methodenaufruf !");
        }
...
         var mytask = new Task(test);
         mytask.Start();
...
```

Anonyme Methode:

```
            var mytask = new Task(delegate() { Console.WriteLine("Anonyme Methode !"); });
            mytask.Start();
```

Verwendung eines Lambda-Ausdrucks:

```
            var mytask = new Task(() => {Console.WriteLine("Lambda-Ausdruck!");});
            mytask.Start();
```

Für welche der drei Varianten Sie sich entscheiden, hängt neben den eigenen Vorlieben sicher auch vom Umfang der Aufgabe ab.

10.5.2 Den Task starten

Wie schon erwähnt, wird bei Verwendung der Methode *Task.Factory.StartNew* der Task automatisch gestartet, weitere Anweisungen sind also nicht erforderlich.

BEISPIEL 10.12: Automatischer Start mit Task.Factory.StartNew

C#

```
...
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
            Console.WriteLine("Start");
```

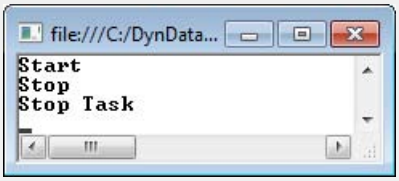
Ab hier läuft bereits der Task, den Rückgabewert der Methode *StartNew* können Sie zum Steuern und Abfragen des Task verwenden:

```
            var t = Task.Factory.StartNew(() =>
            {
                Thread.Sleep(1000);
                Console.WriteLine("Stop Task");
            });
            Console.WriteLine("Stop");
            Console.Read();
        }
...
```

BEISPIEL 10.12: Automatischer Start mit Task.Factory.StartNew

Ergebnis

Das erwartete Ergebnis:



Möchten Sie beim Erzeugen die Art der Task-Verwaltung beeinflussen, können Sie zusätzliche Optionen übergeben.

BEISPIEL 10.13: Verwendung von *TaskCreationOptions.LongRunning*

C#

```
var mytask = Task.Factory.StartNew(MeineMethode,TaskCreationOptions.LongRunning );
```

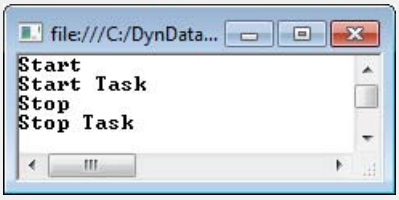
BEISPIEL 10.14: Variante 2 explizites Starten

C#

```
...
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
            Console.WriteLine("Start");
            var t = new Task(() => {
                Thread.Sleep(1000);
                Console.WriteLine("Stop Task");
            });
            Console.WriteLine("Start Task");
            t.Start();
            Console.WriteLine("Stop");
            Console.Read();
        }
...
```

Ergebnis

Die Ausgabe:



10.5.3 Datenübergabe an den Task

Starten Sie einen extra Task ist es vielfach wünschenswert, Daten an diesen zu übergeben. Nichts leichter als dies, definieren Sie einfach einen Parameter (*Object*) für den Lambda-Ausdruck und übergeben Sie die Daten als weiteren Parameter an die *StartNew*-Methode bzw. den Task-Konstruktor.

BEISPIEL 10.15: Übergabe eines Objekts vom Typ *CKontakte*

C#

```
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
```

Instanz der Klasse definieren (Klassendefinition siehe am Ende des Listings):

```
            CKontakte kontakt = new CKontakte {Nachname = "Müller",
                                Vorname = "Heinz", Telefon = "01901234567" };
            Console.WriteLine("Start");
```

Hier definieren wir einen Eingabeparameter für die Lambda-Funktion:

```
            var t = new Task((myData) =>
            {
```

Wir müssen den Übergabewert typisieren:

```
                CKontakte myLocalObject = (CKontakte) myData;
                Thread.Sleep(1000);
```

Verwenden der Daten:

```
                Console.WriteLine(myLocalObject.Nachname);
                Console.WriteLine(myLocalObject.Vorname);
                Console.WriteLine(myLocalObject.Telefon);
                Console.WriteLine("Stop Task");
```

Hier findet die Übergabe der Daten an den Task statt:

```
            }, kontakt);
            Console.WriteLine("Start Task");
            t.Start();
            Console.WriteLine("Stop");
            Console.Read();
        }
    }
```

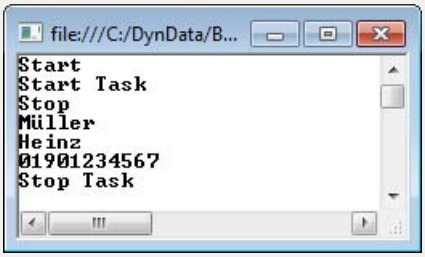
Die Klassendefinition:

```
    public class CKontakte
    {
```

BEISPIEL 10.15: Übergabe eines Objekts vom Typ *CKontakte*

```
        public String Nachname;
        public String Vorname;
        public String Telefon;
    }
```

Ergebnis



HINWEIS: Möchten Sie Daten aus dem Task zurückgeben, sollten Sie sich noch etwas gedulden, ab Seite 520 finden Sie die Lösung.

10.5.4 Wie warte ich auf das Ende des Task?

Alles hat ein Ende und – wie die Wurst – kann auch der Task mit zwei "Enden" aufwarten:

- das reguläre Ende nach der kompletten Abarbeitung der Routine und
- das weniger schöne Ende durch eine auftretende Exception.

In diesem Abschnitt wollen wir uns mit der normalen Abarbeitung des Task beschäftigen, Gründe für das Warten bieten sich einige an, so das Synchronisieren mit anderen Tasks oder die Auswertung von Berechnungs- oder Abfrageergebnissen aus dem Task.

Die naheliegendste Lösung bietet die *Wait*-Methode, die Sie über die aktuelle *Task*-Instanz aufrufen können.

BEISPIEL 10.16: Warten auf das Task-Ende

C#

```
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
            Console.WriteLine("Start");
            var t = new Task(() =>
            {
                Thread.Sleep(5000);
                Console.WriteLine("Stop Task");
            });
```

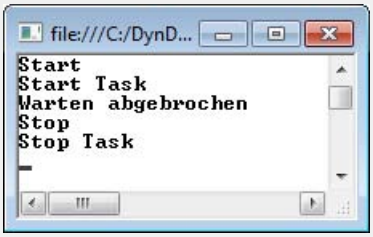
BEISPIEL 10.16: Warten auf das Task-Ende

```
            Console.WriteLine("Start Task");
            t.Start();
```

Hier warten wir 2 Sekunden auf das Taskende (die Zeit wird nicht reichen, da wir den Task 5 Sekunden lang am Leben lassen) und werten das Ergebnis der Warterei aus:

```
            if (t.Wait(2000))
                Console.WriteLine("Warten erfolgreich");
            else
                Console.WriteLine("Warten abgebrochen");
            Console.WriteLine("Stop");
            Console.ReadKey();
...
```

Ergebnis



Wer das vorhergehende Kapitel aufmerksam gelesen hat, wird sich sicher an entsprechende Konstrukte bei den Threads erinnern.

HINWEIS: Werden aus einem Task heraus weitere Tasks gestartet, wartet die *Wait*-Methode auch auf das Ende dieser Child-Tasks.

Alternativ kann es auch vorkommen, dass Sie nicht nur auf einen, sondern auf mehrere Tasks warten müssen, z.B. um diese miteinander zu synchronisieren. In diesem Fall nutzen Sie bitte die Methode *WaitAll*.

HINWEIS: Soll nur auf einen Task aus einer Liste von Tasks gewartet werden, nutzen Sie die *WaitAny*-Methode.

BEISPIEL 10.17: Verwendung von *WaitAll*

C#

```
...
            var t1 = Task.Factory.StartNew(() => {Thread.Sleep(3000);});
            var t2 = Task.Factory.StartNew(() => {Thread.Sleep(5000);});
            var t3 = Task.Factory.StartNew(() => {Thread.Sleep(2000);});
            Task.WaitAll(t1, t2, t3);
...
```

10.5.5 Tasks mit Rückgabewerten

Was liegt näher, als einen Task für umfangreiche Berechnungen oder Auswertungen zu verwenden? Doch damit tritt auch die Frage in den Vordergrund, wie ich die ermittelten Werte an das aufrufende Programm zurückgeben kann. Für diese Aufgabenstellung bietet sich die *Task*<>-Klasse an. Diese bietet über die *Result*-Eigenschaft einen typisierten Zugriff auf den Rückgabewert des Tasks. Statt vieler Worte zunächst ein einfaches Beispiel.

BEISPIEL 10.18: Rückgabe eines Integerwertes

C#

```
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
```

Datentyp für die Rückgabe definieren:

```
            Task<int> t = new Task<int>(() => {
                    Thread.Sleep(5000);
```

Hier erfolgt das Zuweisen des Rückgabewertes:

```
                    return 27;
                });
            Console.WriteLine("Start Task");
```

Task starten:

```
            t.Start();
```

Ergebnis auswerten:

```
            Console.WriteLine("Rückgabewert = " + t.Result.ToString());
        }
...
```

Mancher wird sich jetzt vielleicht fragen, warum wir direkt nach dem Start bereits auf das Ergebnis zugreifen können. Die Antwort ist recht einfach: Jeder Zugriff auf *Result* synchronisiert die Ausführung des Task, d.h. es wird gewartet, bis der Task beendet ist.

Alternativ können Sie beispielsweise auch mit der *IsCompleted*-Eigenschaft abfragen, ob das Ergebnis schon vorliegt und dann entsprechend auf den Ergebniswert zugreifen (quasi Polling).

HINWEIS: Wie es auch besser geht, zeigt der Abschnitt 10.6, wo Sie mit der Erweiterungsmethode *ContinueWith* auf das Ende des Tasks reagieren können.

Dass sich die Rückgabewerte nicht auf einfache Datentypen beschränken, soll das folgende Beispiel zeigen, bei dem eine Collection von *Files*-Objekten im Task abgerufen und später aufgelistet wird.

BEISPIEL 10.19: Task mit Rückgabewert (Collection)

C#

```
...
using System.IO;
using System.Threading;
using System.Threading.Tasks;
...
        static void Main(string[] args)
        {
```

Hier wir der Task definiert:

```
            Task<FileInfo[]> t1 = new Task<FileInfo[]>(() => {
                    DirectoryInfo di = new DirectoryInfo(@"c:\");
                    return di.GetFiles();
                });
```

Optional könnten Sie auch rekursiv durch die Verzeichnisse iterieren, dies dauert je nach Plattenfüllstand entsprechend lange und dürfte einen Hintergrundtask in jedem Fall rechtfertigen.

```
            Console.WriteLine("Start Task");
```

Task starten:

```
            t1.Start();
```

Rückgabewerte abrufen (hier verläuft die Verarbeitung synchron):

```
            Console.WriteLine("Rückgabewerte");
            foreach (FileInfo f in t1.Result)
            {
                Console.WriteLine(f.FullName);
            }
            Console.ReadKey();
...
```

HINWEIS: Natürlich können Sie auch mit den Funktionsergebnissen aus mehreren Tasks Berechnungen realisieren, in diesen Fällen wird so lange gewartet, bis alle erforderlichen Funktionsergebnisse vorliegen.

BEISPIEL 10.20: Wer es etwas einfacher mag, kann auch die folgende Lösung mit der Methode *Task.Factory.StartNew* wählen.

C#

```
...
            var t1 = Task.Factory.StartNew(() =>
            {
                DirectoryInfo di = new DirectoryInfo(@"c:\");
                return di.GetFiles();
            });
            Console.WriteLine("Rückgabewerte");
```

BEISPIEL 10.20: Wer es etwas einfacher mag, kann auch die folgende Lösung mit der Methode *Task.Factory.StartNew* wählen.

```
            foreach (FileInfo f in t1.Result)
                Console.WriteLine(f.FullName);
...
```

HINWEIS: In diesem Fall kann die Typangabe weggelassen werden.

Doch was ist, wenn Sie mehr als einen Rückgabewert haben? Auch hier hilft Ihnen die obige Lösung weiter, erzeugen Sie einfach ein neues Objekt mit den Rückgabewerten als Eigenschaften.

BEISPIEL 10.21: Mehrere (drei) Rückgabewerte definieren

C#

Indem Sie neue Objekte erzeugen, ist es problemlos möglich, jede Art von Informationen aus den Tasks zurückzugeben:

```
var t1 = Task.Factory.StartNew(() =>
{
    return new { Zeit = System.DateTime.Now,
                 Wert = 4711, Zeichenkette = "abcdefg" };
});
```

Sie sollten sich auch die Zeit nehmen, Bezeichner für die einzelnen Eigenschaften zu vergeben, die spätere Lesbarkeit beim Zugriff auf die Rückgabewerte entschädigt Sie für den Mehraufwand.

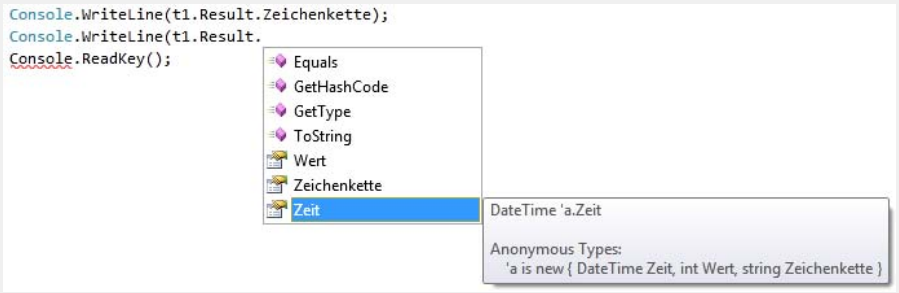
```
Console.WriteLine("Rückgabewerte");
```

Rückgabewerte anzeigen:

```
Console.WriteLine(t1.Result.Wert);
Console.WriteLine(t1.Result.Zeichenkette);
Console.WriteLine(t1.Result.Zeit);
Console.ReadKey();
```

Ergebnis

Wer jetzt glaubt, auf Intellisense und Typprüfung beim späteren Zugriff auf die Ergebnisse verzichten zu müssen, wird schnell eines besseren belehrt:



Damit dürfte auch die Rückgabe umfangreicher (Datenbank-)Informationen kein Problem darstellen. Womit wir bereits bei zwei recht diffizilen Themen angekommen sind: dem Abbrechen der Taskverarbeitung und einer entsprechenden Fehlerbehandlung.

10.5.6 Die Verarbeitung abbrechen

Auch wenn es sich nicht um den Regelfall handelt, zählt doch der Abbruch eines Task zu den gelegentlich anfallenden Aufgaben des Programmierers. Grundsätzlich bestehen die folgenden Möglichkeiten zum Task-Abbruch:

- Sie beenden den Task, indem Sie aus der Task-Logik heraus die *return*-Anweisung aufrufen.
- Sie verwenden einen von der *CancellationTokenSource*-Klasse bereitgestellten *CancellationToken*, um eine Exception auszulösen, die Sie natürlich im weiteren Verlauf abfangen müssen[1].

HINWEIS: Die zweite Variante ist auch für den Abbruch von *Parallel.For* bzw. *Parallel.ForEach*-Statements anwendbar.

Abbruch per return

Diese Variante dürfte Sie nicht vor große Herausforderungen stellen, es genügt, wenn Sie innerhalb der Task-Logik die *return*-Anweisung aufrufen. Beachten Sie allerdings, dass der Rückgabestatus des Tasks (*IsCompleted*-Eigenschaft) in diesem Fall ein reguläres *true* liefert.

BEISPIEL 10.22: Reguläres Abbrechen eines Tasks (nach 10 Schleifendurchläufen)

C#

```
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
...
        private void button3_Click(object sender, EventArgs e)
        {
            listBox1.Items.Insert(0, "Task starten ...");
```

Task definieren und starten:

```
            myTask = Task.Factory.StartNew(() =>
            {
                int i = 0;
                while (true)
                {
```

Hier steigen wir später aus:

```
                    if (i == 10) return;
```

[1] Das Auslösen einer Exception soll bereits andeuten, dass es sich eben nicht um den regulären Abbruch des Tasks handelt.

BEISPIEL 10.22: Reguläres Abbrechen eines Tasks (nach 10 Schleifendurchläufen)

Ein paar Lebenszeichen an das Consolenfenster senden:

```
                Debug.WriteLine("Wert von i = " + i.ToString()); // Test only
                Thread.Sleep(500);                    // etwas Aktivität simulieren
                i++;
            }
        });
```

Hier warten wir auf den Task[1], um später den Status auszulesen:

```
        myTask.Wait();
        listBox1.Items.Add("Status: " + myTask.Status.ToString());
        listBox1.Items.Add("IsCanceled: " + myTask.IsCanceled);
        listBox1.Items.Add("IsCompleted: " + myTask.IsCompleted);
    }
```

Ergebnis

Wie zu erwarten, lässt sich über den Task-Status nicht der Grund für die Beendigung in Erfahrung bringen. Eventuell könnten Sie eine eigene Statusvariable für diesen Fall über *return* zurückgeben.

Abbruch per CancellationToken

Damit sind wir auch schon bei der Standard-Lösung für das Abbrechen von Tasks (und parallelen Threads) angelangt. Wie das folgende Beispiel zeigt, handelt es sich nicht um eine Lösung, die Sie mit zwei Codezeilen realisieren können, dafür bietet diese Variante jedoch auch Ihre Vorteile.

Das Grundprinzip basiert auf der Verwendung eines *CancellationTokenSource*-Objekts. Dieses stellt einen *CancellationToken* bereit, den Sie in einem oder auch in mehreren Tasks verwenden können. Der Token selbst bietet zum einen die Eigenschaft *IsCancellationRequested*, die Auskunft darüber gibt, ob das besitzende *CancellationTokenSource*-Objekt einen Abbruch per *Cancel*-Methode eingeleitet hat. "Eingeleitet" bedeutet nichts mehr und nichts weniger als die Aufforderung zum Abbruch, ob Sie diesen im Task wirklich einleiten ist Ihre Entscheidung.

Ist die Eigenschaft *IsCancellationRequested* auf *true* gesetzt, können Sie noch Code zum "Aufräumen" realisieren, bevor Sie final einen Exception auslösen, indem Sie die Token-Methode *ThrowIfCancellationRequested* aufrufen.

Nachfolgend sind Sie dafür verantwortlich, diesen Fehler abzufangen. Dies kann im Rahmen eines *Wait*-Statements oder bei der Auswertung eines Task-Rückgabewertes erfolgen (siehe folgendes Beispiel).

[1] Eigentlich recht sinnlos, da dadurch die Oberfläche blockiert wird.

BEISPIEL 10.23: Task per Exception abbrechen

C#

```
...
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
...
    public partial class Form1 : Form
    {
```

Zunächst brauchen wir eine zentrale *CancellationTokenSource* und einen *CancellationToken*:

```
        CancellationTokenSource cts;
        CancellationToken token;
        Task myTask;

        public Form1()
        {
            InitializeComponent();
```

CancellationTokenSource und *CancellationToken* initialisieren:

```
            cts = new CancellationTokenSource();
            token = cts.Token;
        }
```

Hier kommen wir zum Starten des eigentlichen Tasks:

```
        private void button1_Click(object sender, EventArgs e)
        {
            listBox1.Items.Insert(0, "Task starten ...");
            myTask = Task.Factory.StartNew(() =>
            {
                int i = 0;
                while (true)
                {
```

Etwas Aktivität im Ausgabefenster realisieren:

```
                    Debug.WriteLine("Wert von i = " + i.ToString()); // Test only
```

Hier reagieren wir auf eine Abbruchanforderung:

```
                    if (token.IsCancellationRequested)
                    {
                        // Hier noch sinnvolles Beenden möglich
```

Hier wir eine Exception ausgelöst, der Task endet damit:

```
                        token.ThrowIfCancellationRequested();
                    }
                    Thread.Sleep(500);                      // etwas Aktivität simulieren
                    i++;
                }
```

BEISPIEL 10.23: Task per Exception abbrechen

Wichtig: Der Token muss an den Task beim Erstellen übergeben werden:

```
        }, token);
    }
```

Die Routine zum Abbruch des Task:

```
    private void button2_Click(object sender, EventArgs e)
    {
        listBox1.Items.Insert(0, "Abbruch anfordern ...");
```

Hier wird die Aufforderung ausgelöst:

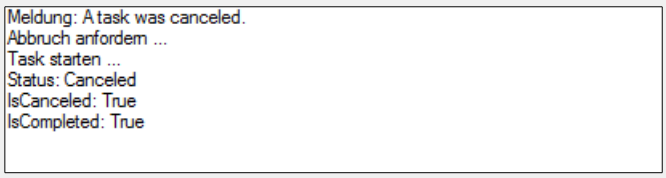
```
        cts.Cancel();
```

Im Rahmen eines *Wait*-Statements fangen wir den Fehler auf und zeigen die Fehlermeldung an:

```
        try
        {
            myTask.Wait();
        }
        catch (AggregateException ex)
        {
            foreach (var v in ex.InnerExceptions)
               listBox1.Items.Insert(0, "Meldung: " + v.Message);
        }
        listBox1.Items.Add("Status: " + myTask.Status.ToString());
        listBox1.Items.Add("IsCanceled: " + myTask.IsCanceled);
        listBox1.Items.Add("IsCompleted: " + myTask.IsCompleted);
    }
```

Ergebnis

Das Ergebnis in der Anzeige:



Wie Sie sehen, meldet zwar *IsCompleted* einen erfolgreichen Abschluss des Tasks, aber sowohl *Status* als auch *IsCanceled* zeigen den Abbruch des Tasks an.

10.5.7 Fehlerbehandlung

Wie Sie nach Lektüre des vorhergehenden Abschnitts vielleicht schon erkannt haben, wurde in diesem Zusammenhang auch gleich en passant die Fehlerbehandlung bei Task-Exceptions aufgezeigt.

Da Tasks auch verschachtelt realisiert werden können[1], werden ausgelöste Exceptions in einem *AggregateException* "verpackt". Die Ausführung wird an den aufrufenden Thread zurückgegeben und dieser hat die Aufgabe, den Fehler entsprechend zu behandeln. Allerdings stellt sich hier die Frage, zu welchem Zeitpunkt dies geschehen soll, läuft doch der Task in der Regel asynchron, sodass Sie kein *try-catch*-Statement sinnvoll aufbauen können.

Der Grundansatz ist deshalb die Fragestellung, zu welchem Zeitpunkt eine Fehlerbehandlung überhaupt einen Sinn macht. Allgemein wird dies immer dann der Fall sein, wenn auf den Task bzw. dessen Ergebnisse gewartet wird. Hier sollten Sie sich die Methoden *Wait*, *WaitAll* und *WaitAny* sowie die Abfrage der *Result*-Eigenschaft vormerken. Diese Anweisungen kapseln Sie in einen *try-catch*-Block und lesen mit einer *foreach*-Schleife die *InnerExceptions* aus.

BEISPIEL 10.24: (Auszug aus dem vorhergehenden Beispiel)

C#

```
try
{
    myTask.Wait();
}
catch (AggregateException ex)
{
    foreach (var v in ex.InnerExceptions)
       listBox1.Items.Insert(0, "Meldung: " + v.Message);
}
```

HINWEIS: Beachten Sie, dass Sie die Fehlermeldungen in der Visual Studio-IDE in diesem Fall ausschalten müssen, andernfalls unterbricht der Debugger bereits in der Task-Routine und zeigt die auftretende Exception an.

10.5.8 Weitere Eigenschaften

Bevor wir zu weiteren wichtigen Themen übergehen, wollen wir noch einen Blick auf einige weitere Eigenschaften und Optionen von Tasks werfen.

Task-Id

Da findet sich zum einen die *Id*, die eine eindeutige Identifikation jedes Tasks (zum Beispiel im Debugger) ermöglicht.

HINWEIS: Die Zuweisung des Wertes erfolgt bei der ersten Verwendung der Eigenschaft, spätere Programmaufrufe können also andere Werte zurückgeben.

Mehr zu diese Thematik finden Sie im Abschnitt 10.9, wo wir uns speziell dem Debugging widmen wollen.

[1] Ein Umstand, der sicher viele gute, aber leider auch unlesbare Programme hervorbringen wird ...

Status

Diese Eigenschaft bietet detaillierte Informationen über den aktuellen Status des Tasks:

Wert	Beschreibung
Created *WaitingForActivation* *WaitingToRun*	Task wurde erzeugt, wird jedoch noch nicht ausgeführt (verschiedene Zwischenzustände werden unterschieden).
Running	Der Task läuft gerade.
WaitingForChildrenToComplete	Der Task selbst ist mit der Verarbeitung fertig, es wird noch auf Child-Tasks gewartet.
RanToCompletion	Der Task wurde regulär beendet.
Canceled	Der Task wurde per Token abgebrochen.
Faulted	Im Task ist eine Exception aufgetreten, die Verarbeitung wurde abgebrochen.

IsCanceled, IsCompleted, IsFaulted

Über diese Eigenschaften können Sie sich ebenfalls ein Bild vom aktuellen Status des Task machen. Wie schon erwähnt, ist *IsCompleted* auch dann *true*, wenn der Task abgebrochen wurde. *IsCanceled* wird auf *true* gesetzt wenn der Task per Token abgebrochen wird, *IsFaulted* ist bei jedem anderen auftretendem Fehler *true*.

TaskCreationOptions

Mit den *TaskCreationOptions* können Sie beim Erstellen des Tasks dessen Verhalten bzw. die Steuerung durch den Task-Scheduler beeinflussen.

Wert	Beschreibung
None	Keine weitere Spezifikation.
PreferFairness	Beeinflusst den Scheduler: Eher erstellte Aufgaben werden auch eher verarbeitet, gleiches gilt im Umkehrschluss für später erstellte Aufgaben.
LongRunning	Spezifiziert einen Task als lang laufende Aufgabe.
AttachedToParent	Der spezifizierte Task wird als Child dem aktuellen Task zugeordnet.

10.6 Zugriff auf das User-Interface

Auch bei Verwendung der TPL stehen Sie vor dem Problem, dass ein direkter Zugriff aus dem jeweiligen Thread/Task auf Elemente der Programmoberfläche (Controls) ausgeschlossen ist.

Zunächst können wir Sie guten Gewissens auf den Abschnitt 9.4 verweisen, wo sowohl auf die Problematik selbst als auch auf die Lösung mittels *Invoke/BeginInvoke* eingegangen wird. Die dort

vorgestellte Lösung sollten Sie immer dann verwenden, wenn es sich um Aktualisierungen der Oberfläche während der Laufzeit des Threads bzw. Tasks handelt.

Im Folgenden stellen wir Ihnen ergänzend zwei Lösungen vor, mit denen Sie auf das Ende des Tasks reagieren können und aus dem Task heraus auf die Oberfläche zugreifen können.

10.6.1 Task-Ende und Zugriff auf die Oberfläche

Eine wohl häufig auftretende Aufgabe ist die Reaktion auf das Task-Ende. Meist geht es darum, Ergebnisse auf den Bildschirm zu bringen oder die Ergebnisse des Tasks anderweitig zu verarbeiten.

In diesem Fall bietet sich für den Task die Erweiterungsmethode *ContinueWith* an, der Sie neben einem Delegate/Lambda-Ausdruck auch ein *TaskScheduler*-Objekt übergeben können. Der Clou daran: Sie können das *TaskScheduler*-Objekt des aktuellen UI-Threads über die Methode *FromCurrentSynchronizationContext* abrufen und verwenden. Damit finden alle Ausgaben aus *ContinueWith* heraus im Context des Userinterface statt, umständliche *Invoke*-Aufrufe entfallen gänzlich.

BEISPIEL 10.25: Warten auf Task-Ende

C#

Abrufen aller Dateien aus dem Verzeichnis[1] "C:\Windows\System32" in einem Hintergrund-Task und Anzeige in einer *ListBox*, wenn alle Dateien abgerufen wurden.

```
using System.IO;
using System.Threading;
using System.Threading.Tasks;
...
        private void button1_Click(object sender, EventArgs e)
        {
```

Wir ermitteln den Task-Scheduler des UI-Threads:

```
            var uiSched = TaskScheduler.FromCurrentSynchronizationContext();
```

Neuen Task definieren und starten:

```
            Task.Factory.StartNew(() =>
                {
                    DirectoryInfo di = new DirectoryInfo(@"c:\windows\system32");
                    Thread.Sleep(3000);                     // etwas Aktivität simulieren
```

Rückgabewert ist die Liste der Dateinamen:

```
                    return di.GetFiles() ;
```

Die folgende Aktion wird ausgeführt, wenn der Task beendet ist (der Parameter entspricht dem Rückgabewert des Tasks):

```
                }).ContinueWith(t =>  {
                    listBox1.Items.Clear();
```

[1] Eventuell anpassen ...

BEISPIEL 10.25: Warten auf Task-Ende

Task-Rückgabewert auswerten:

```
                foreach (FileInfo f in t.Result)
                {
                    listBox1.Items.Add(f.FullName);
                }
```

Achtung: Wir müssen noch den passenden Task-Scheduler angeben:

```
            }, uiSched);
        }
```

HINWEIS: Die Programmausführung wird zunächst unabhängig vom Task-Ende weitergeführt, die Aktion in *ContinueWith* wird später asynchron aufgerufen.

Wer diese Version mit den *Invoke*-Orgien bei den Threads vergleicht, dürfte sich schnell mit dieser Lösung anfreunden können. Doch ein Wunsch bleibt vermutlich immer noch offen: Zugriff auf das Userinterface aus dem Task heraus.

10.6.2 Zugriff auf das UI aus dem Task heraus

Neben der oben genannten Lösung können Sie auch einen neuen Task erstellen, der im Context eines anderen Threads (und hier denken wir natürlich gleich an die Oberfläche) läuft. Genau diese Möglichkeit bietet sich auch für das Aktualisieren der Oberfläche aus dem Task heraus an.

BEISPIEL 10.26: Aktualisieren einer *ListBox* aus einem laufenden Task heraus

C#

```
using System.Threading;
using System.Threading.Tasks;
...
        private void button2_Click(object sender, EventArgs e)
        {
```

Zunächst ermitteln wir wieder das *TaskScheduler*-Objekt für die Oberfläche:

```
            var uiSched = TaskScheduler.FromCurrentSynchronizationContext();
```

Ganz nebenbei benötigen wir noch einen *CancellationToken* (eigentlich benötigen wir diesen nicht, die Parameterliste für das Erzeugen eines Task mit vorgegebenem TaskScheduler erfordert jedoch diese Angabe):

```
            var token = Task.Factory.CancellationToken;
```

Hier starten wir unseren Arbeits-Task:

```
            Task.Factory.StartNew(() =>
            {
                for (int i = 0; i <10; i++)
                {
```

BEISPIEL 10.26: Aktualisieren einer *ListBox* aus einem laufenden Task heraus

Wir wollen die Oberfläche bei jedem Schleifendurchlauf aktualisieren und erzeugen dazu einen neuen Anzeige-Task, der im Kontext der Oberfläche läuft:

```
                Task.Factory.StartNew(() =>
                {
                    listBox1.Items.Insert(0, "Wert von i = " + i.ToString());
                }, token, TaskCreationOptions.None, uiSched);
```

Hier legen wir eine Verschnaufpause im Arbeits-Task ein:

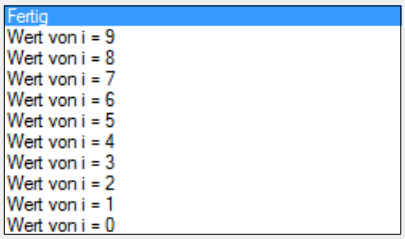
```
                Thread.Sleep(1000);                    // etwas Aktivität simulieren
            }
```

Ist der Arbeits-Task beendet, zeigen wir noch eine Abschlussmeldung an:

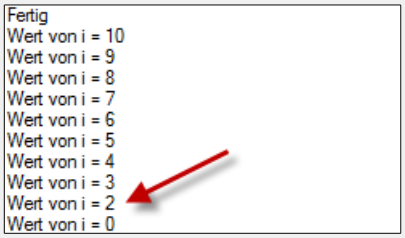
```
            }).ContinueWith(t =>
            {
               listBox1.Items.Insert(0,"Fertig");
            }, uiSched);
        }
...
```

Ergebnis

Ein Start des Programms wird Ihnen zyklisch die Zwischenergebnisse in der *ListBox* anzeigen, die Oberfläche ist in dieser Zeit nicht blockiert:



Verschieben Sie ruhig einmal die *Thread.Sleep*-Anweisung vor das Erzeugen des Anzeige-Task. Sie dürften wahrscheinlich ebenfalls ein ähnliches Ergebnis in der *ListBox* erhalten:



Nanu, da stimmt doch etwas nicht! Die Lösung für dieses "Durcheinander": Da der Anzeige-Task nicht synchron mit dem Arbeits-Task läuft, ist auch der Zugriff auf die Schleifenvariable i nicht "threadsicher", die Anzeige erfolgt mal mit dem alten und mal mit dem neuen Schlei-

BEISPIEL 10.26: Aktualisieren einer *ListBox* aus einem laufenden Task heraus

fenwert, je nach Laufzeitdauer des Anzeige-Threads. Beachten Sie dieses bei Verwendung der o.g. Lösung in Ihren Programmen. Eventuell müssen Sie sogar ihren Anzeige-Task mit *Wait* entsprechend synchronisieren:

```
            Task.Factory.StartNew(() =>
            {
                for (int i = 0; i <10; i++)
                {
                    Thread.Sleep(1000);                        // etwas Aktivität simulieren
                    var t = Task.Factory.StartNew(() =>
                    {
                        listBox1.Items.Insert(0, "Wert von i = " + i.ToString());
                    }, token, TaskCreationOptions.None, uiSched);
                    t.Wait();
```

Jetzt wartet der Arbeits-Thread auf den Anzeige-Thread!

```
                }
            }).ContinueWith(t =>
...
```

Mit den Lösungen aus den beiden vorhergehenden Abschnitten dürften einige Herzenswünsche des Task-Programmierers in Erfüllung gegangen sein[1].

10.7 Weitere Datenstrukturen im Überblick

Nachdem wir uns durch den Hauptteil der TPL gewühlt haben, wollen wir abschließend noch einen kurzen Blick auf die TPL-Datenstrukturen werfen, die entweder die Arbeit des Programmieres erleichtern sollen oder einfach nur Nachfolger für bereits bekannte Strukturen aus der Thread-Programmierung sind.

HINWEIS: Leider ist es uns aus Platzgründen nicht möglich, dieses Thema in gebotener Ausführlichkeit zu behandeln, wir verweisen Sie stattdessen auf die entsprechende Online-Hilfe.

10.7.1 Threadsichere Collections

Sicher ist Ihnen nach Lektüre der letzten beiden Kapitel der teilweise recht hohe Aufwand beim threadübergreifenen Zugriff auf Variablen und Collections aufgefallen. Ist ein Schreibzugriff erforderlich, müssen die entsprechenden Anweisungen akribisch gesichert und damit threadsicher gemacht werden (z.B. mit *lock*). Um dem Programmierer dieses Arbeit abzunehmen und gleich-

[1] Leider werden diese Lösungen aber auch dazu beitragen, dass einige Entwickler wieder entnervt auf den Bildschirm starren und Ihre eigenen Programme nicht mehr verstehen.

zeitig ein zeitoptimales Ergebnis zu erzielen, bietet die TPL einige threadsichere Collections für Standardaufgaben an (Namespace *System.Collections.Concurrent*):

- *BlockingCollection*
- *ConcurrentBag*
- *ConcurrentDictionary*
- *ConcurrentQueue*
- *ConcurrentStack*

Die Verwendung dürfte sich weitgehend aus den Bezeichnern ergeben, weitere Informationen finden Sie im Kapitel 5.

10.7.2 Primitive für die Threadsynchronisation

Wie schon im Abschnitt 9.3 ab Seite 445 beschrieben, werden für die Steuerung von Threads/Tasks diverse Möglichkeiten bereitgehalten, um zum Beispiel Threads miteinander zu synchronisieren oder Ressourcen gemeinsam zu nutzen. Auch hier bietet die TPL einige Erweiteruzngen, die wir lediglich aufzählen wollen:

- *Barrier,*
- *CountdownEvent,*
- *ManualResetEventSlim,*
- *SemaphoreSlim,*
- *SpinLock* und *SpinWait*.

10.8 Parallel LINQ (PLINQ)

Nachdem Sie mittlerweile vermutlich alles parallelisiert haben, was sich nur ansatzweise dafür anbietet, fällt Ihr Blick sicher früher oder später auch auf Collections und damit auf die sich in diesem Zusammenhang anbietenden LINQ-Anweisungen. Bevor Sie jetzt anfangen, hier mühsam etwas zu parallelisieren, sollten Sie sich mit dem bereist verfügbaren Werkzeugen besser vertraut machen und einen Blick auf PLINQ bzw. Parallel-LINQ werfen.

HINWEIS: Bevor wir uns an dieser Stelle wiederholen, möchten wir Sie zurück ins Kapitel 6 (Abschnitt 6.3.9) schicken, wo wir uns der Thematik "PLINQ" bereits angenommen haben.

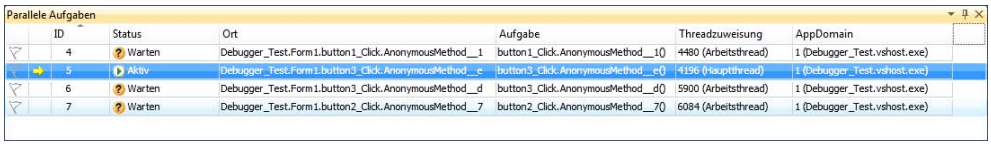
10.9 Die Parallel Diagnostic Tools

Sicher ist Ihnen bei einigen Beispielen der letzten beiden Kapitel schon der Gedanke gekommen, dass Ihre Programme im Chaos versinken und Sie den Überblick restlos verlieren könnten. Doch keine Angst, nicht nur die Sprachen bzw. das Framework haben sich weiterentwickelt, sondern auch die Visual Studio IDE und dort speziell die Tools zu Überwachung paralleler Abläufe.

Unter dem Menüpunkt *Debuggen|Fenster* finden Sie unter anderem die Menüpunkte *Parallele Aufgaben* und *Parallele Stapel*. Beide sollen Sie bei der Analyse von Tasks unterstützen. Ein weiteres, bei der Analyse von asynchronen Programmen recht interessantes Feature ist die IntelliTrace-Funktion, die es Ihnen ermöglicht, Informationen über zurückliegende Ereignisse/Ausnahmen etc. zu sammeln und zu einem späteren Zeitpunkt auszuwerten.

10.9.1 Fenster für parallele Aufgaben

Wie bereits erwähnt, erreichen Sie dieses Fenster über den Menüpunkt *Debuggen|Fenster*. Haben Sie die Programmausführung über einen Breakpoint oder zum Beispiel durch eine auftretende Exception unterbrochen, können Sie sich in diesem Fenster über die aktuell laufenden (oder auch pausierenden) Tasks informieren:



Angezeigt werden neben der Task-Id der aktuelle Status, der Ort sowie die ausgeführte Aufgabe (Lambda-Ausdrücke werden zu anonymen Methoden). Für die bessere Analyse bietet dieses Fenster die Möglichkeit, den aktuellen Task oder alle weiteren angezeigten Task "einzufrieren".

Interessant sind die Statusinformationen, bei denen folgende Zustände unterschieden werden:

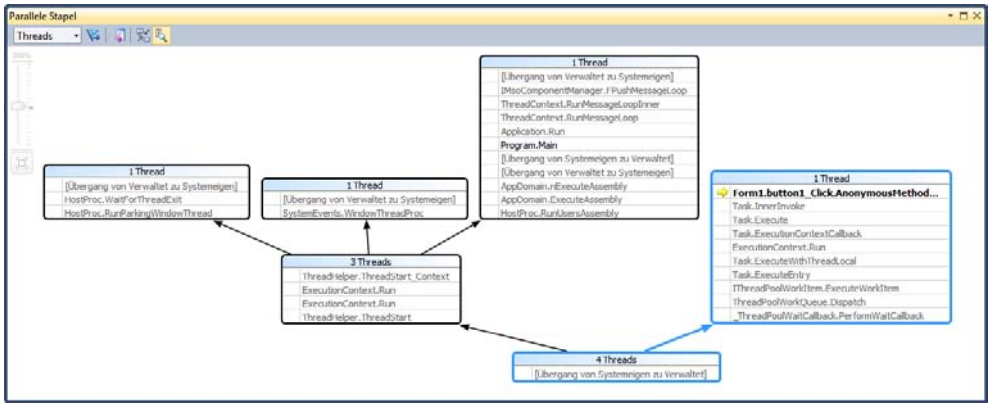
- **Scheduled**
  Der Task wurde lediglich erstellt aber noch nicht ausgeführt. Er verfügt über keinen Call Stack, zugeordnete Threads oder sonstige Informationen.
- **Running**
  Dieser Task wurde vor der Unterbrechung durch den Debugger ausgeführt.
- **Waiting**
  Der betreffende Task ist blockiert, da er auf einen anderen Task, das Aufheben einer Sperre oder auf ein Signal wartet.
- **Deadlocked**
  Der Thread dieses Tasks befindet sich in einem Deadlock mit einem anderen Thread.

Bei den beiden letzten Zuständen werden Zusatzinformationen angezeigt, wenn Sie die Maus in die Statuszelle bewegen.

HINWEIS: Sie können zusätzlich auch die Spalte *Parent* einblenden, hier können Sie Informationen über die Hierarchie voneinander abhängiger Tasks erhalten.

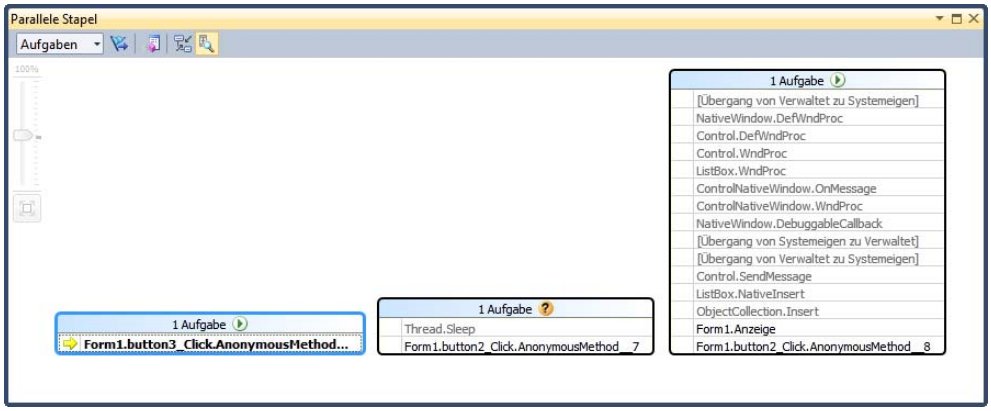
10.9.2 Fenster für parallele Stacks

Das Fenster für parallele Stacks zeigt Ihnen Call Stack-Informationen für alle Threads der aktuellen Anwendung, dazu gehören auch die standardmäßig für die Anwendung laufenden Threads, die unabhängig von den User-Threads erzeugt werden:



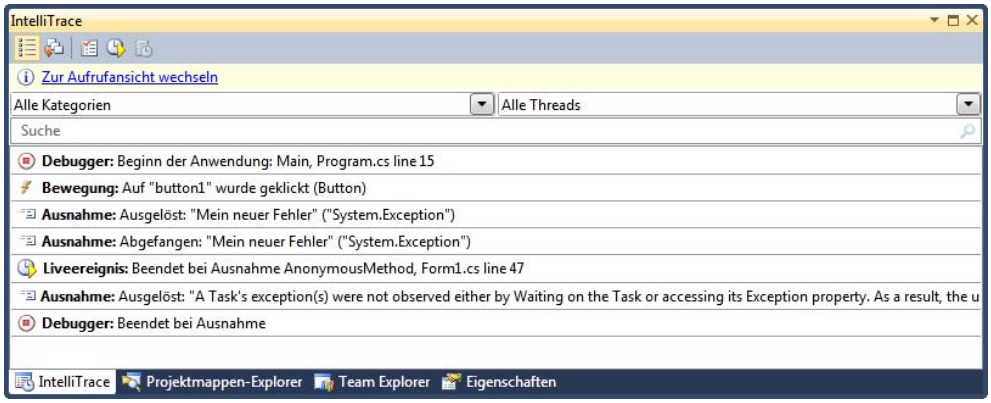
In obiger Abbildung finden Sie lediglich einen selbst definierten Thread (in diesem Fall handelt es sich um einen Task basierend auf einem Lambda-Ausdruck) auf der rechten Seite. Aus den fetten (blauen) Linien können Sie den Aufruf-Pfad des aktuellen Threads erkennen. Der gelbe Pfeil zeigt den gerade aktive Stack Frame, Sie können diesen wechseln, indem Sie auf eine andere Methode klicken. Dabei ist es egal, ob es sich um den aktuellen oder um einen anderen Thread handelt.

Alternativ können Sie über die *ComboBox* am linken oberen Rand auch in die Task-Ansicht wechseln, hier finden sich dann lediglich die von Ihnen definierten und gestarteten Tasks mit den jeweiligen Statusinformationen wieder.



10.9.3 IntelliTrace

Eine bereits mit Visual Studio 2010 eingeführte Funktion ist das sogenannte IntelliTrace, eine Möglichkeit, nach der Ausführung des Programms Ausnahmen und deren Kontext näher zu untersuchen (siehe folgende Abbildung).



Klicken Sie auf einen der angezeigten Einträge, wird Ihnen die entsprechende Codezeile Ihrer Anwendung angezeigt.

HINWEIS: Alternativ können Sie über *Tools|Optionen|IntelliTrace* (Rubrik "Ereignisse und Aufrufinformationen") weitere Informationen aufzeichnen lassen. Dies kann allerdings die Performance Ihrer Anwendung negativ beeinflussen.

10.10 Praxisbeispiel: Spieltrieb – Version 2

Auch am Ende dieses Kapitels wollen wir es nicht versäumen, die besprochenen Sprachkonstrukte mehr oder weniger vollständig an einem etwas komplexeren Beispiel vorzuführen. Wer bereits das vorherige Kapitel 9 durchgearbeitet hat, wird sicher auch auf das Multi-Threading-Beispiel am Schluss des Kapitels gestoßen sein. Anhand einer einfachen Verkehrs-Simulation wurden dort diverse thread-spezifische Funktionen erörtert.

An dieser Stelle wollen wir uns dieses Beispiel noch einmal vorknöpfen und mit Hilfe der neuen Task-spezifischen Sprachelemente neu programmieren. Auf entsprechende Abbildungen und Erklärungen zur Oberfläche können wir an dieser Stelle verzichten, dies wurde alles beibehalten, lediglich die Logik hinter den Kulissen musste teilweise komplett umgebaut werden.

10.10.1 Aufgabenstellung

Zur Erinnerung noch einmal die Kurzbeschreibung: Ausgehend von der Benutzereingabe sollen in einem Lager Kisten hinzugefügt werden. Diese werden von drei LKWs zu einem Schiff transportiert. Ist das Schiff mit fünf Kisten beladen, transportiert es diese weiter, löscht die Ladung und

kehrt zurück, um erneut Kisten zu laden usw. Einzige Ausnahme: Ist mindestens eine Kiste an Bord und nach 15 Sekunden keine neue Ladung zu erwarten, legt das Schiff trotzdem ab. Ist das Schiff nicht da, müssen die LKWs natürlich am Hafen warten.

Auch diesmal werden wir das "Problem" auf drei Klassen aufteilen:

- *LKW*
- *Schiff*
- *Controller* (Lager-Controller)

Innerhalb der Klassen werden wir ausgiebig von den neuen Tasks Gebrauch machen, doch im Unterschied zur vorhergehenden Version sind diese Tasks komplett innerhalb der Klassen gekapselt, ein Zugriff von außen ist weder möglich noch nötig.

10.10.2 Global-Klasse

Auch in diesem Fall können/wollen wir nicht auf eine globale statische Klasse verzichten, die uns eine übergreifende Kommunikation zwischen den einzelnen o.g. Klassen ermöglicht.

```
using System.Threading;
using System.Threading.Tasks;  // Neu ab .NET 4.0
...
    static class Global
    {
```

Die Instanzen unserer obigen Klassen:

```
        public static cController Controller;
        public static Schiff Schiff;
        public static cLKW[] LKWs = new cLKW[3];
```

Den Zugriff auf die Oberfläche ermöglicht uns die folgende Variable, in der wir die Instanz von *Form1* speichern werden:

```
        public static Form1 myForm;
```

Den *TaskScheduler* des UI speichern wir ebenfalls, so können wir aus den Tasks heraus auf die Oberfläche zugreifen, ohne explizit *Invoke* zu verwenden:

```
        public static TaskScheduler uiSched;
```

Für die Vollständigkeit aller Parameter beim Erstellen einiger Tasks benötigen wir noch einen *CancellationToken*:

```
        private static CancellationToken token = Task.Factory.CancellationToken;
```

Für die threadsichere Anzeige der Statusmeldungen bzw. zum Aktualisieren der Oberfläche nutzen wir einen extra Task, der mit dem *TaskScheduler* des UI gestartet wird:

```
        public static void Anzeige(string msg)
        {
```

Task erzeugen und starten:

```
            Task.Factory.StartNew(() =>
            {
                if (msg != "")
                    myForm.listBox1.Items.Insert(0, System.DateTime.Now + " : " + msg);
                else
                {
                    myForm.label3.Text = Global.Schiff.Ladung.ToString();
                    myForm.label4.Text = Global.Schiff.Transportiert.ToString();
                    myForm.label6.Text = Global.Controller.Lagerbestand.ToString();
                    myForm.label10.Text = Global.LKWs[0].kmStand.ToString();
                    myForm.label11.Text = Global.LKWs[1].kmStand.ToString();
                    myForm.label12.Text = Global.LKWs[2].kmStand.ToString();
                    myForm.pictureBox2.Image = Global.LKWs[0].Image;
                    myForm.pictureBox3.Image = Global.LKWs[1].Image;
                    myForm.pictureBox4.Image = Global.LKWs[2].Image;
                    myForm.pictureBox1.Left = Global.Schiff.pos;
                    myForm.pictureBox2.Left = Global.LKWs[0].Pos;
                    myForm.pictureBox3.Left = Global.LKWs[1].Pos;
                    myForm.pictureBox4.Left = Global.LKWs[2].Pos;
                }
            }, token, TaskCreationOptions.None, uiSched);
        }
    }
}
```

HINWEIS: Mehr zu dieser Technik siehe Abschnitt 10.6, ab Seite 528.

10.10.3 Controller-Klasse

Für die übergreifende Steuerung sorgt unser Controller, den wir ebenfalls "runderneuert" haben:

```
using System.Threading;
using System.Threading.Tasks;            // ab .NET 4.0
...
    class cController
    {
```

Der interne Lagerbestand, bevor die Ware von den LKWs abgeholt wird:

```
        public int Lagerbestand;
```

Um keine CPU-Zeit zu verschwenden, warten wir mit einem *AutoResetEvent* bis neue Ware ins Lager gestellt wird:

```
        private AutoResetEvent NeueWareDa = new AutoResetEvent(false);
```

Im Grunde handelt es sich um so etwas wie ein Flag, auf das mit *Wait* gewartet werden kann. Jeder Aufruf von *Set* setzt das Flag erneut, nach einem *Wait*-Duchlauf wird das Flag automatisch zurückgesetzt.

Die vom UI aufzurufenden Methode, wenn neue Ware ins Lager gestellt wird:

```
public void NeueWare()
{
    Lagerbestand++;
```

Hier geben wir die Sperre frei, der Controller kann wieder arbeiten:

```
    NeueWareDa.Set();
}
```

Unser "Hauptprogramm":

```
public cController()
{
```

Task mit Endlosschleife erzeugen:

```
    Task.Factory.StartNew(() =>
    {
        Global.Anzeige("");
        Global.Anzeige("Controller: Gestartet");
        int i = 0;
        while (true)
        {
```

Ist der Lagerbestand größer null, wird versucht, die LKWs zu beladen:

```
            while (Lagerbestand > 0)
            {
                // Ist der LKW nicht unterwegs, dann beladen
                if (Global.LKWs[i].ImLager)
                {
                    Global.Anzeige("Controller: Lkw beladen");
                    Global.LKWs[i].Abtransport();
                }
```

Falls der gewünschte LKW nicht vorhanden ist, versuchen wir es mit dem nächsten, etwas Pause zwischendurch verhindert unsinnige CPU-Belastung[1]:

```
                i++;
                if (i > 2) i = 0;
                Thread.Sleep(100);
            }
```

Falls keine Ware da ist gehen wir hier in den Dämmerschlaf über, erst wenn neue Ware herein-kommt, wird diese Sperre wieder freigegeben:

```
            NeueWareDa.WaitOne();
```

[1] Wie im echten Leben ...

```
                }
            });
        }
    }
}
```

10.10.4 LKW-Klasse

```
using System.Threading.Tasks;  // neu ab .Net 4.0
...
    class cLKW
    {
        public int Nr;
        public int Pos;
        public int kmStand;
        public bool ImLager;
        private Image Img1;
        private Image Img2;
        public Image Image;
```

LKW initialisieren:

```
        public cLKW(int nr)
        {
            Nr = nr;
            kmStand = 0;
            Pos = 280;
            ImLager = true;
            Img1 = Properties.Resources.Auto1;
            Img2 = Properties.Resources.Auto2;
            Image = Img1;
        }
```

Diese Methode wird durch den Controller aufgerufen und gewährleistet sowohl Hinfahrt/Rückfahrt, als auch die reibungslose Übergabe der Ware an das Schiff:

```
        public void Abtransport()
        {
            ImLager = false;
```

Den Lagerbestand verringern:

```
            Global.Controller.Lagerbestand--;
```

Neuer Task:

```
            Task.Factory.StartNew(() =>
            {
                Thread.Sleep(200); // Ladezeit
                Global.Anzeige("Lkw_" + Nr.ToString() + ": Fahrtbeginn");
                Image = Img1;
```

Hinfahrt:

```
                for (Pos = 280; Pos > 0; Pos -= 2)
                {
                    Thread.Sleep(30);
                    kmStand++;
                    Global.Anzeige("");
                }
```

Ankunft im Hafen, auf Schiff warten:

```
                Global.Schiff.Beladen();
```

Im Gegensatz zur vorhergehenden Variante ist die obige Methode blockierend, wenn kein Schiff vorhanden ist. Ein Zugriff auf den Status des Schiffs (interner Task) ist deshalb nicht mehr nötig, wir können uns auf das Wesentliche beschränken.

Rückfahrt:

```
                Image = Img2;
                for (Pos = 0; Pos < 280; Pos += 2)
                {
                    Thread.Sleep(20);
                    kmStand++;
                    Global.Anzeige("");
                }
                ImLager = true;
                Global.Anzeige("Lkw_" + Nr.ToString() + ": Fahrtende");
            });
        }

    }
}
```

10.10.5 Schiff-Klasse

"Ein Schiff wird kommen ...", was im Schlager einfach so dahingesagt/gesungen wird, erfordert dann doch etwas mehr Aufwand, wie das folgende Listing zeigt:

```
using System.Threading;
using System.Threading.Tasks;
...
    class Schiff
    {
        public int Ladung = 0;
        public int Transportiert = 0;
        public int pos = 200;
```

In dieser Klasse benötigen wir zwei Sperren, *ImHafen* wird manuell gesteuert (Abfahrt und Ankunft des Schiffs, *NeueWareDa* löst den Beladevorgang aus, wenn ein LKW angekommen ist):

```
        public ManualResetEvent ImHafen = new ManualResetEvent(true);
```

```
        private AutoResetEvent NeueWareDa = new AutoResetEvent(false);
```

Die zentrale Transport-Routine:

```
        public void Transport()
        {
```

Die Sperre manuell setzen:

```
            ImHafen.Reset();
```

Abfahrt:

```
            Global.Anzeige("Schiff: Abfahrt");
            for (pos = 200; pos > 0; pos -= 2)
            {
                Thread.Sleep(100);
                Global.Anzeige("");
            }
```

Im *Hafen2* die Ladung löschen:

```
            Global.Anzeige("Schiff: Ladung löschen");
            while (Ladung > 0)
            {
                Thread.Sleep(400);
```

Da nur noch ein *Task* auf *Ladung* bzw. *Transportiert* zugreifen kann, können wir jetzt auf einen *Monitor/lock*-Abschnitt verzichten:

```
                Ladung--;
                Transportiert++;
                Global.Anzeige("");
            }
```

Die Rückfahrt:

```
            for (pos = 0; pos < 200; pos += 2)
            {
                Thread.Sleep(60);
                Global.Anzeige("");
            }
            Global.Anzeige("Schiff: Warten ...");
```

Sperre manuell zurücksetzen:

```
            ImHafen.Set();
        }
```

Die blockierende Methode *Beladen:*

```
        public void Beladen()
        {
```

Hier warten wir auf das Schiff:

```
            ImHafen.WaitOne();
            Ladung++;
            Global.Anzeige("Schiff: Beladen (" + Ladung.ToString() + ")");
```

Sperre für den Haupt-Task freigeben:

```
            NeueWareDa.Set();
        }
```

Der Konstruktor:

```
        public Schiff()
        {
```

Neuen Task erstellen:

```
            Task.Factory.StartNew(() =>
            {
```

Eine Endlosschleife:

```
                while (true)
                {
```

Wenn keine Ware geladen ist warten wir, bis neue Ladung per LKW heran transportiert wird (unbegrenzt):

```
                    if (Ladung == 0)
                        NeueWareDa.WaitOne();
```

Wenn bereits Ware geladen ist, warten wir auf Ladung, jedoch maximal 15 Sekunden:

```
                    else
                    {
```

Wurde neue Ware angeliefert, Transport wenn fünf Stück geladen sind:

```
                        if (NeueWareDa.WaitOne(15000))   // Neuer Artikel
                           if (Ladung == 5) Transport();
```

Andernfalls ist nur ein Timeout aufgetreten, wir fahren im Zweifel auch mit einem Stück weg:

```
                        else
                        {
                            Global.Anzeige("Schiff: Zeit abgelaufen");
                            Transport();
                        }
                    }
                }
            });
        }
    }
}
```

10.10.6 Oberfläche

Schon beim Vorgängerbeispiel war hier wenig Code erforderlich, jetzt aber ist es noch weniger, was der Funktionalität sicherlich keinen Abbruch tut und für noch mehr Übersicht sorgen dürfte:

```
using System.Threading;
using System.Threading.Tasks;
...
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
```

Unsere Objekte initialisieren:

```
            Global.uiSched = TaskScheduler.FromCurrentSynchronizationContext();
            Global.LKWs[0] = new cLKW(0);
            Global.LKWs[1] = new cLKW(1);
            Global.LKWs[2] = new cLKW(2);
            Global.Schiff = new Schiff();
            Global.Controller = new cController();
```

Achtung: Die Referenz für das Formular speichern:

```
            Global.myForm = this;
        }
```

HINWEIS: Vergessen Sie in diesem Zusammenhang bitte nicht, dass die Oberflächensteuerelemente, die von der Klasse *Global* angesprochen werden, auch als *public* markiert sein müssen, andernfalls haben Sie von dort keinen Zugriff!

Mit einem Klick auf die "+"-Schaltfläche fügen Sie neue Waren zum Lager hinzu. Dies sollte für hektische Betriebsamkeit beim Controller sorgen, nachfolgend bekommen die LKWs etwas zu tun und, last but not least, auch das Schiff.

```
        private void button1_Click(object sender, EventArgs e)
        {
            Global.Controller.NeueWare();
        }
```

Das Schließen des Formulars gestaltet sich recht unkompliziert, das Beenden der Task wollen wir ohne extra Aufwand bewerkstelligen:

```
        private void button2_Click(object sender, EventArgs e)
        {
            Close();
        }
    }
```

HINWEIS: Wer die Task "sauber" beenden will, sollte sich mit dem Abschnitt 10.5.6 vertraut machen.

Damit sind wir auch schon am Ende unseres Beispiels und auch des Kapitels angelangt. Wer die beiden Beispiele miteinander vergleicht wird feststellen, dass die neue Version wesentlich weniger Ressourcen verbraucht und auch eleganter programmiert ist. Allerdings ist die Grafikausgabe jetzt etwas ruckeliger geworden, da wir auf die Rechenzeitverteilung per Task-Scheduler keinen direkten Einfluss haben, aber dies war ja auch nicht Gegenstand unserer Betrachtungen.

Kapitel 11

Fehlersuche und Behandlung

Ein Programm, das auf Anhieb 100%-ig funktioniert, gibt es so gut wie nicht. Besonders tückisch sind die logischen Fehler, die weder zur Entwurfs- noch zur Laufzeit auftreten und die sich lediglich in einer fehlerhaften Funktion des Programms (z.B. falsche Rechenergebnisse, mysteriöse Abstürze) äußern. Für uns Grund genug, im ersten Abschnitt dieses Kapitels den in Visual Studio integrierten Debugger genauer unter die Lupe zu nehmen.

Während das Debugging nur im Stadium der Programmentwicklung von Interesse ist, sollte eine "wasserdichte" Fehlerbehandlung allen zur Laufzeit nur denkbaren Missgeschicken mit geeigneten Mitteln begegnen. Dieser Problematik widmen sich die weiteren Abschnitte des Kapitels.

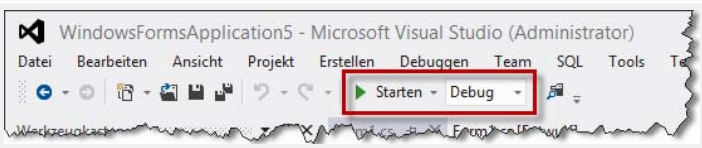
11.1 Der Debugger

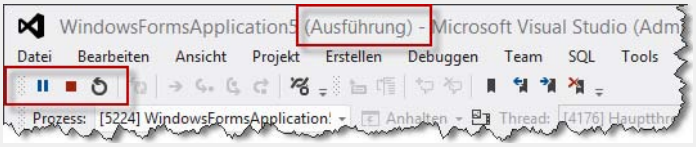
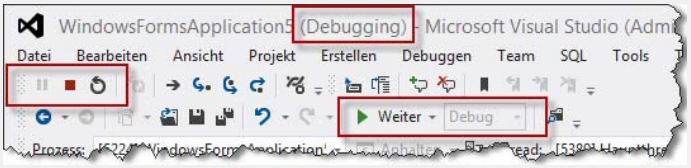
Der in Visual Studio 2012 integrierte Debugger ist ein recht komplexes Werkzeug, ausgestattet mit einer Fülle von Features und Fenstern. Zumindest den Umgang mit den elementaren Debug-Funktionen (auf diese wollen wir uns im Folgenden beschränken) sollte der Programmierer im Schlaf beherrschen, spart er sich doch dadurch viel Zeit und Nerven.

11.1.1 Allgemeine Beschreibung

Beim Debugging spielt der aktuelle Modus des Programms (Entwurfsmodus, Ausführungsmodus oder Unterbrechungs- bzw. Debuggingmodus) eine entscheidende Rolle.

Die Beschriftung der Visual Studio-Titelleiste informiert Sie, in welchem der drei Modi sich das Programm gegenwärtig befindet. Gleichzeitig zeigt Ihnen der Zustand der Schaltflächen *Starten* , *Unterbrechen* und *Beenden* die möglichen Alternativen zum Wechseln des aktuellen Modus.

Modus	Titelleiste und Schaltflächen
Entwurfsmodus	

Modus	Titelleiste und Schaltflächen
Ausführungsmodus	
Unterbrechungsmodus	

Das eigentliche Debugging wird in der Regel im Unterbrechungsmodus vollzogen. Die Zeile, an welcher die Programmausführung unterbrochen ist, erscheint dabei gelb hinterlegt:

HINWEIS: Sie dürfen auch im Debugging-Modus Ihren Code editieren und danach das Programm fortsetzen. Allerdings kann nicht alles Mögliche editiert werden, so können Sie zum Beispiel private Felder oder Methoden hinzufügen, entfernen dürfen Sie aber keine.

11.1.2 Die wichtigsten Fenster

Über das Menü *Ansicht|Weitere Fenster* bzw. *Debuggen|Fenster* lässt sich eine verwirrende Vielfalt von Fenstern aufrufen, die mit dem Debugging irgendwie im Zusammenhang stehen und die in der Regel am unteren Rand der IDE erscheinen. Die für den Einstieg zunächst wichtigsten Fenster sind:

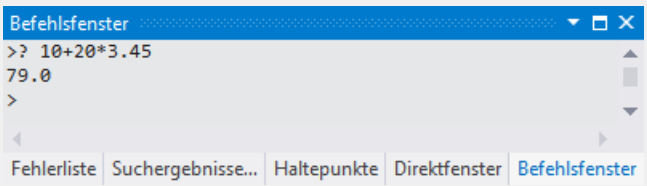
- Befehlsfenster (*Ansicht|Weitere Fenster|Befehlsfenster*)
- Direktfenster (*Debuggen|Fenster|Direkt*)
- Lokalfenster (*Debuggen|Fenster|Lokal*)
- Überwachungsfenster (*Debuggen|Fenster|Überwachen*)
- Autofenster (*Debuggen|Fenster|Auto*)
- Ausgabefenster (*Ansicht |Ausgabe* oder *Debuggen|Fenster| Ausgabe*)

HINWEIS: Einige dieser Fenster werden nur im Ausführungs- bzw. Unterbrechungs-Modus angeboten!

Befehlsfenster

Unabhängig vom aktuellen Programm können Sie in diesem Fenster nach Belieben mit C#-Anweisungen experimentieren.

BEISPIEL 11.1: Ausführen einer Rechenaufgabe (10+20*3,45=79,0) im unmittelbaren Modus



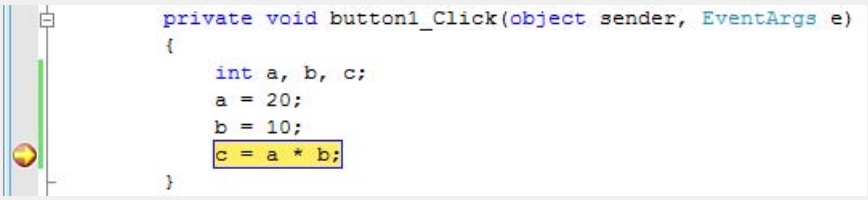
HINWEIS: Nützlich ist das Befehlsfenster unter anderem auch für das schnelle Ausprobieren diverser Optionen (zum Beispiel *Format*-Anweisungen o.ä.).

Direktfenster

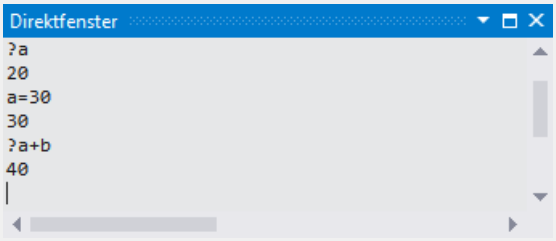
Das Direktfenster erlaubt es Ihnen, sich während des Debugging-Prozesses Werte von Variablen oder Ausdrücken ausgeben zu lassen, oder sogar diese Werte zu ändern.

HINWEIS: Das Programm muss sich dazu im Unterbrechungsmodus befinden, den Sie z.B. durch Setzen eines Haltepunktes erzwingen können.

BEISPIEL 11.2: Sie erzeugen im Entwurfsmodus einen Haltepunkt (einfach durch Klick auf die linke graue Randleiste) und gelangen somit nach Programmstart in den Unterbrechungsmodus:

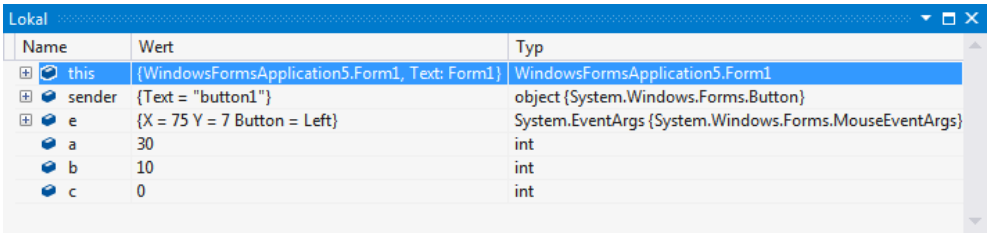


Im Direktfenster können Sie nun verschiedene Operationen durchführen:



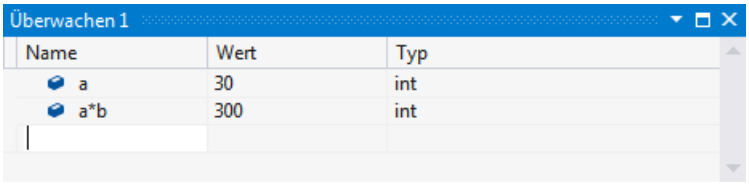
Lokal-Fenster

Hier lassen sich – vorausgesetzt das Programm befindet sich im Unterbrechungsmodus – Namen, Wert und Typ aller aktuellen Variablen und Objekte anzeigen:



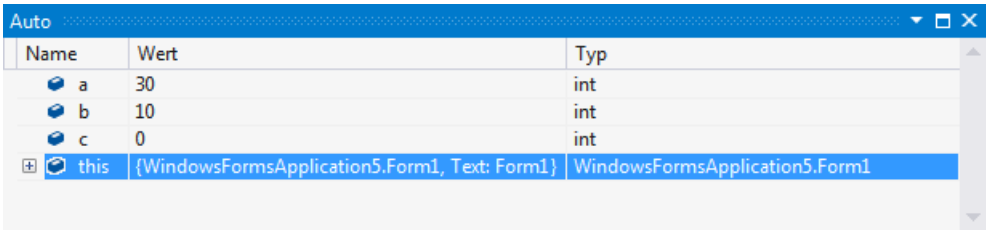
Überwachungsfenster

Der Wert bestimmter Variablen bzw. Ausdrücke (Watch-Expressions) kann hier während der Laufzeit überwacht werden. Welche Variablen oder Ausdrücke Sie überwachen wollen, bestimmen Sie durch die Eingabe in die Spalte *Name*:



Autofenster

Dieses Fenster zeigt Ihnen alle Variablen der vorhergehenden und der Zeile an, in welcher sich der Haltemodus gerade befindet.



Ausgabefenster

In diesem Fenster werden Ihnen neben diversen Statusmeldungen von Visual Studio auch die Ausgaben des *Debug*- bzw. *Trace*-Objekts angezeigt (siehe 11.2 Arbeiten mit Debug und Trace).

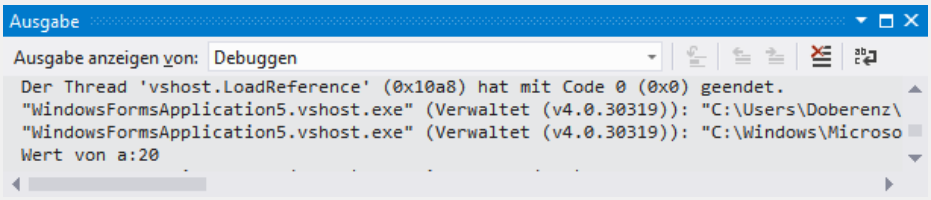
BEISPIEL 11.3: Anzeige von Werten im Ausgabefenster

C#
```
using System.Diagnostics;
...
   int a,b,c;
   a = 20;
   Debug.WriteLine("Wert von a:" + a);
```

BEISPIEL 11.3: Anzeige von Werten im Ausgabefenster

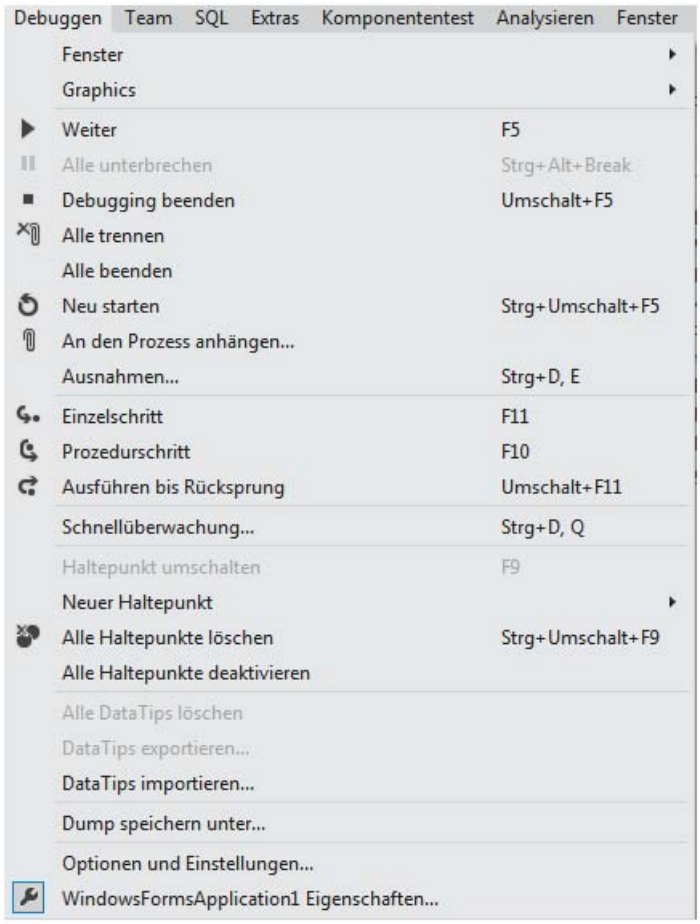
Ergebnis

Das dazugehörige Ausgabefenster:



11.1.3 Debugging-Optionen

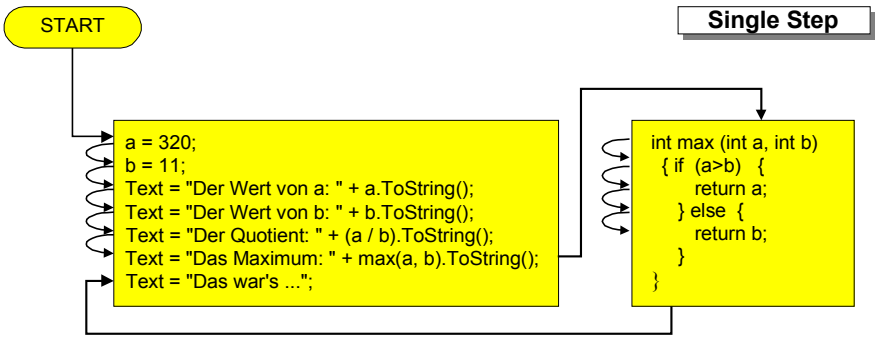
An der Anzahl der Einträge im Menü *Debuggen* erkennen Sie bereits die Vielfalt der Debugging-Möglichkeiten:



Viele Fehler lassen sich nur durch schrittweises Abarbeiten des Quelltextes eingrenzen, Sie können dabei zwischen mehreren Varianten wählen:

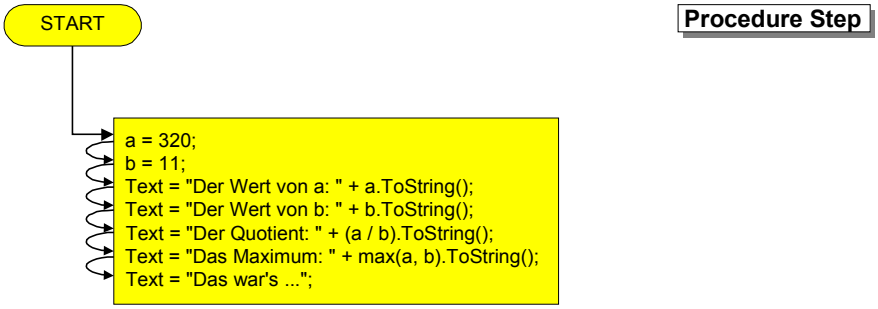
Einzelschritt-Modus

Im *Single-Step*-(Einzelschritt-)Modus durchlaufen Sie das Programm (mit *F11*) zeilenweise. Nach der Ausführung einer Anweisung liegt der Unterbrechungsmodus vor.



Prozedurschritt-Modus

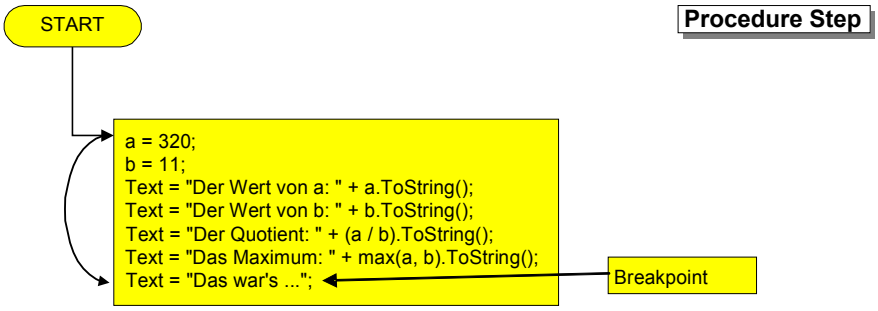
Bei umfangreicheren Programmen ist das zeilenweise Abarbeiten recht mühsam. Abhilfe schafft hier der *Procedure-Step*-(Prozedurschritt-)Modus (mit *F10*). Zwar wird nach wie vor jede einzelne Zeile verarbeitet, innerhalb von Unterprogrammen wird allerdings nicht mehr angehalten.



Breakpoints

Am komfortabelsten dürfte wohl das Setzen von *Breakpoints* (Haltepunkten) sein. Sie können dazu den Kursor in die entsprechende Anweisung platzieren und die Taste *F9* drücken.

Starten Sie das Programm wie gewohnt, die Ausführung wird an genau dieser Stelle unterbrochen. Auch das Setzen mehrerer Haltepunkte ist möglich.



11.1.4 Praktisches Debugging am Beispiel

Um die einzelnen Betriebsarten zu testen, können Sie natürlich eine beliebige Anwendung nehmen. Die folgenden Ausführungen beziehen sich auf eine einfache WinForm-Demo, die sich an obige Abbildungen anlehnt und lediglich aus einem Formular besteht.

BEISPIEL 11.4: Einfache Debugging-Demo

C#

Nach dem Programmstart werden die folgenden Anweisungen ausgeführt:

```
private void Form1_Load(object sender, System.EventArgs e)
{
      int a,b,c;
      a = 320;
      b = 11;
      for (c = 0; c < 15; c++)
      {
        b = b + a + 10;
        b = b - 12 * a;
      }
}
```

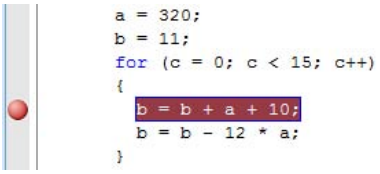
Die Funktion *Max*:

```
private int max(int a, int b)
{
   if (b > a )
      return b;
   else
      return a;
}
```

HINWEIS: Obiges Programm hat, wie Sie sehen, keinerlei sinnvolle Funktion – hier geht es lediglich um die Erläuterung des Grundprinzips.

Haltepunkte setzen

Einen Breakpoint setzen Sie entweder über das Menü *Debuggen|Neuer Haltepunkt* oder Sie klicken einfach mit der Maus auf den linken grauen Rand des Codefensters, und es erscheint ein fetter dunkelroter Punkt, auch die gesamte Zeile wird markiert. Genauso einfach lassen sich die Haltepunkte wieder entfernen – Sie brauchen nur darauf zu klicken.

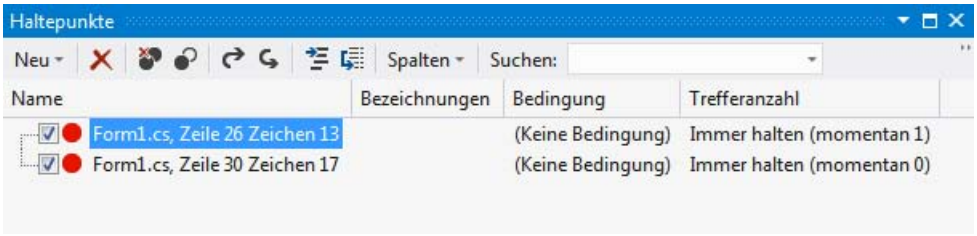


```
a = 320;
b = 11;
for (c = 0; c < 15; c++)
{
  b = b + a + 10;
  b = b - 12 * a;
}
```

Nach dem Programmstart (*F5*) werden alle Anweisungen bis **vor** die Breakpoint-Zeile ausgeführt. Anschließend können Sie schrittweise mit *F11* (Einzelschritt) bzw. *F10* (Prozedurschritt) fortfahren. Natürlich lassen sich auch mehrere Breakpoints setzen.

Eine sehr praktikable Möglichkeit ist das Setzen von Haltepunkten nicht zur Entwurfs-, sondern erst zur Laufzeit. Sie starten dazu Ihr Programm ganz normal mit *F5*. Nachdem z.B. das Eröffnungsformular erschienen ist, holen Sie das entsprechende Codefenster nach vorne und setzen den oder die Haltepunkt(e). Nun klicken Sie z.B. auf einen Button, und das Programm setzt die Ausführung bis zum Haltepunkt fort.

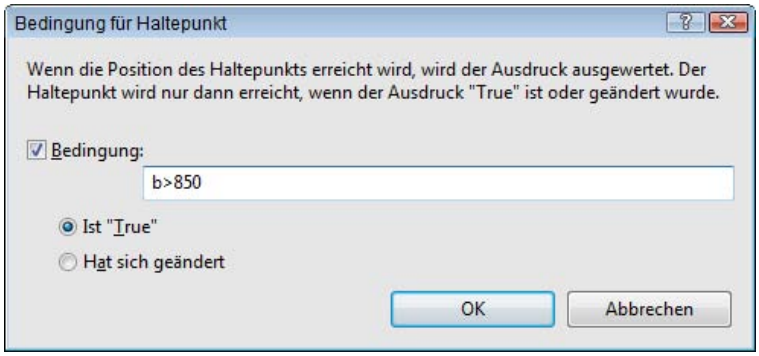
Besonders dann, wenn Sie mehrere Breakpoints gesetzt haben, sind Sie für eine Übersicht dankbar. Wählen Sie dazu das Menü *Debuggen|Fenster|Haltepunkte*. Es erscheint die Liste aller Haltepunkte.



HINWEIS: Um zu einem bestimmten Haltepunkt zu springen, klicken Sie einfach doppelt auf den Eintrag, der Kursor springt automatisch zur gewünschten Stelle.

Abbruchbedingung setzen

Jeder Haltepunkt kann mit einer Abbruchbedingung verknüpft werden. Öffnen Sie zunächst die Liste der Haltepunkte und klicken Sie dann mit der rechten Maustaste auf den Breakpoint, dem Sie eine Bedingung zuordnen möchten. Im PopUp-Menü wählen Sie *Eigenschaften*. In das sich öffnende Dialogfenster wird von Ihnen z.B. die Bedingung *b > 850* eingetragen und mit *OK* bestätigt:

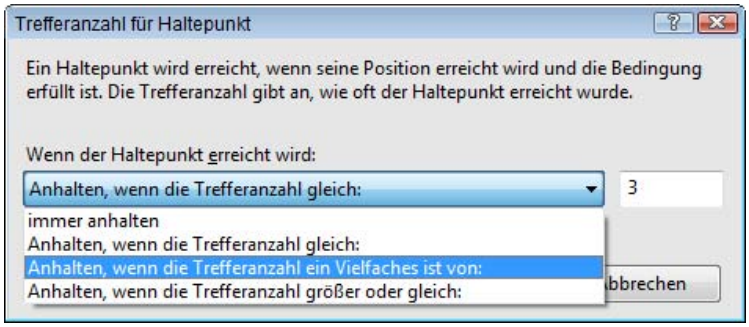


Wenn Sie jetzt unser Demoprogramm normal starten (*F5*), dürfte der Debugger erst dann die Programmausführung anhalten, wenn *c* einen Wert größer zwei hat (*b* ist zu diesem Zeitpunkt größer als *850*).

HINWEIS: Alternativ können Sie den Breakpoint auch setzen wenn sich eine Bedingung ändert.

Trefferanzahl verwenden

Reichen Ihnen die bisherigen Bedingungen nicht aus, können Sie zusätzlich auch eine Trefferzahl als Stopp-Bedingung festlegen. Unabhängig davon, ob Sie bereits eine Abbruchbedingung gesetzt haben, kann hier das Erreichen des Breakpoints bei der Programmausführung berücksichtigt werden. Welche Bedingungen zulässig sind, zeigt die folgende Abbildung:

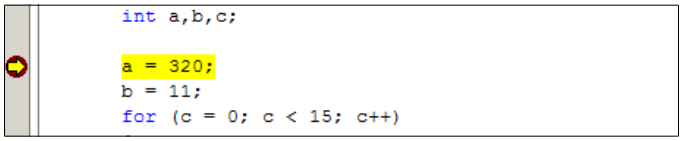


Nach obiger Einstellung und bei Beibehaltung unserer Abbruchbedingung stoppt der Debugger, wenn die Variable *c* den Wert fünf hat (zwei Durchläufe für die Abbruchbedingung und drei für die Trefferanzahl).

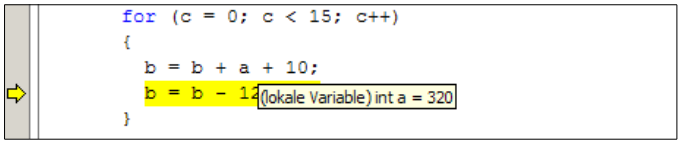
Einzelschritt-Modus

In C# macht der Einzelschrittmodus nur dann einen Sinn, wenn Sie vorher einen Breakpoint gesetzt haben. Das schrittweise Abarbeiten ab der ersten Zeile ist nicht möglich, denn nach dem Durchlauf der *Main*-Methode geht das Programm in die gewohnte Eingabe-Warteschleife.

Haben Sie einen Breakpoint zum Beispiel in der ersten Zeile des *Form_Load*-Ereignisses gesetzt, können Sie ab hier mit F10 (oder der Schaltfläche) zeilenweise den Programmfortschritt beobachten (gelber Pfeil und gelbe Zeilenmarkierung).



Die Werte einzelner Variablen in der Ausführungsposition können Sie nun per gelber QuickInfo kontrollieren, indem Sie den Mauskursor darauf setzen und einen Moment verweilen:



Für diese wie für alle anderen Betriebsarten des Debuggers gilt, dass Sie über *Shift+F5* (bzw. das Menü *Debuggen|Debugging beenden* oder die Schaltfläche ■) den Debug-Modus verlassen können, um in den normalen Entwurfsmodus zurückzukehren.

Prozedurschritt

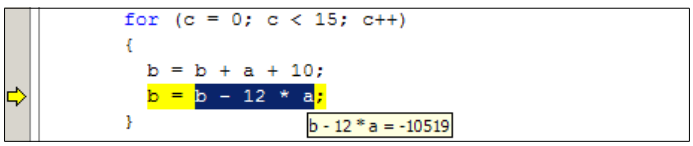
Bei dieser Variante verfahren Sie völlig analog zum Einzelschritt, nur dass Sie diesmal die *F10*-Taste (oder die Schaltfläche) verwenden. Sie werden in unserem Beispiel beobachten, dass bei den Anweisungen innerhalb der *Max*-Methode **nicht** angehalten wird. Den Prozedurschritt werden Sie also nur dann verwenden, wenn es schnell gehen muss und Sie die Fehlerursache außerhalb einer aufgerufenen Methode vermuten.

Ausführen bis Rücksprung

Haben Sie die Möglichkeit des Einzelschritts genutzt und sind Sie in der Methode *Max* angelangt, stellen aber fest, dass der Fehler hier ganz bestimmt nicht zu finden ist, können Sie mit *Shift+F11* (bzw. dem Menüpunkt *Debuggen|Ausführen bis Rücksprung* oder der Schaltfläche) die gesamte Prozedur übergehen und direkt zum eigentlichen Methodenaufruf zurückspringen.

Auswerten von Ausdrücken

Im Einzelschrittmodus arbeiten Sie sich bis zu einer bestimmten Quelltextzeile vor, anschließend markieren Sie den gewünschten Ausdruck und warten auf die Quick-Info:



Insbesondere für mehrfach verschachtelte Funktionsaufrufe und umfangreichere Berechnungen ist dies das Mittel der Wahl, um dem Fehler auf die Schliche zu kommen.

HINWEIS: Ist der Ausdruck zu umfangreich oder möchten Sie die Funktion direkt editieren, können Sie das Befehlsfenster aufrufen, um den Ausdruck auszuwerten. Dabei können Sie auf alle aktiven Variablen zugreifen und diese gegebenenfalls auch ändern.

11.2 Arbeiten mit Debug und Trace

Um das Verhalten einer Anwendung zur Laufzeit zu überprüfen, sind Sie nicht nur auf den in Visual Studio integrierten Debugger angewiesen, sondern Sie können auch selbst Programmcode schreiben, der auf den statischen Klassen *Debug* und *Trace* basiert. Beide Klassen ähneln sich fast so wie ein Ei dem anderen[1].

HINWEIS: Bevor Sie mit *Debug* und *Trace* arbeiten können, müssen Sie den Namespace *System.Diagnostics* einbinden.

11.2.1 Wichtige Methoden von Debug und Trace

Über diese Methoden verfügen sowohl die *Debug*-, als auch die *Trace*-Klasse.

Write, WriteLine, WriteIf und WriteLineIf

Die folgende Tabelle erklärt die vier Methoden, die Sie an fast jeder gewünschten Stelle des Programms unterbringen können.

Methode	Beschreibung
Write	... schreibt Debug-Ausgaben ohne Zeilenumbruch
WriteLine	... schreibt Debug-Ausgaben mit Zeilenumbruch
WriteIf	... schreibt Debug-Ausgaben ohne Zeilenumbruch, wenn eine bestimmte Bedingung erfüllt ist
WriteLineIf	... schreibt Debug-Ausgaben mit Zeilenumbruch, wenn eine bestimmte Bedingung erfüllt ist

Zumindest die Methoden *Write* und *WriteLine* dürften Ihnen von der Klasse *Console* bereits bekannt sein. In unserem Fall wird aber nicht in die Konsole, sondern in das Ausgabe-Fenster von Visual Studio geschrieben, welches sich normalerweise am unteren Rand der IDE befindet (siehe Seite 550). Falls es nicht sichtbar ist, können Sie das Fenster über das Menü *Ansicht|Ausgabe* zur Anzeige bringen. Die folgenden Beispiele sind eine Ergänzung unserer Debugging-Demo von Seite 553.

[1] Auf den "feinen" Unterschied zwischen beiden Klassen werden wir erst an späterer Stelle eingehen.

BEISPIEL 11.5:

C#

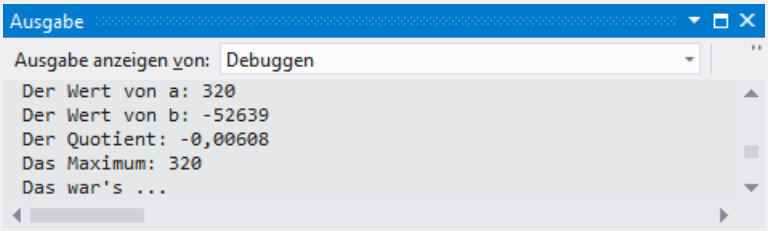
Ergänzen Sie den folgenden Namespace:

```
using System.Diagnostics;
```

Fügen Sie an das Ende des *Form1_Load*-Eventhandlers folgenden Code ein:

```
Debug.WriteLine("Der Wert von a: " + a.ToString());
Debug.WriteLine("Der Wert von b: " + b.ToString());
float f = (float) a / b;
Debug.WriteLine("Der Quotient: " + f.ToString("#0.00000"));
Debug.WriteLine("Das Maximum: " + max(a,b).ToString());
Debug.WriteLine("Das war's ...");
```

Nach Programmstart finden sich im Ausgabefenster folgende Zeilen:



Wie Sie sehen ist – im Unterschied zur Konsolenanwendung – auch eine formatierte Zahlenausgabe möglich.

Alternativ können Sie auch die *Write*-Methode (kein Zeilenumbruch) oder *WriteIf*- bzw. *WriteLineIf*- (Ausgabe, wenn eine anzugebende Bedingung *True* ist) nutzen.

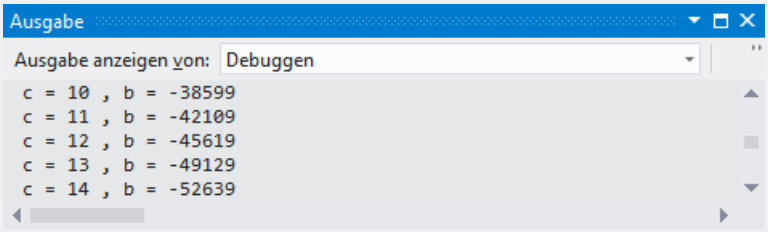
BEISPIEL 11.6: Ausgabe erfolgt nur, wenn die Schleifenvariable c größer gleich 10 ist

C#

Fügen Sie an das Ende der *for*-Schleife im Vorgängerbeispiel folgende Zeile ein:

```
Debug.WriteLineIf(c >= 10, "c = " + c.ToString() + " , b = " + b.ToString() );
```

Das Ausgabefenster:



IndentLevel, Indent, IndentSize und UnIndent

Zur Strukturierung der Ausgabe können Sie verschiedene *Indent*-Methoden verwenden:

Methode/Eigenschaft	Beschreibung
Indent()	... erhöht die aktuelle Einzugsebene um Eins
UnIndent()	... verringert die aktuelle Einzugsebene um Eins
IndentLevel	... liest oder schreibt die Einzugsebene
IndentSize	... liest oder schreibt die Anzahl Leerzeichen eines Einzugs

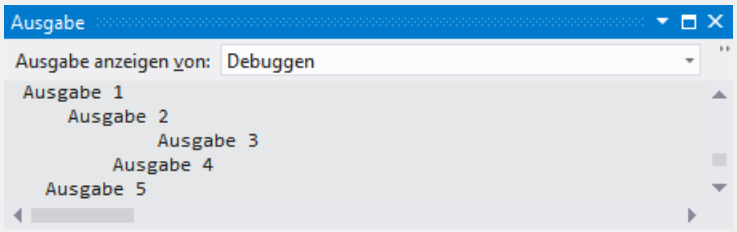
Das folgende Beispiel soll für Klarheit sorgen:

BEISPIEL 11.7: Demonstration verschiedener Texteinzüge

C#

```
using System.Diagnostics;
...
Debug.WriteLine("Ausgabe 1");
Debug.Indent();
Debug.WriteLine("Ausgabe 2");
Debug.IndentLevel = 3;
Debug.WriteLine("Ausgabe 3");
Debug.Unindent();
Debug.WriteLine("Ausgabe 4");
Debug.IndentSize = 2;
Debug.IndentLevel = 1;
Debug.WriteLine("Ausgabe 5");
```

Ergebnis



Assert

Diese Methode erlaubt es, bestimmte Bedingungen (Annahmen) zu prüfen. Falls die Annahme (Assert) falsch ist, wird eine Fehlermeldung angezeigt.

BEISPIEL 11.8: Meldung bei Verletzung einer Annahme

C#

```
using System.Diagnostics;
...
   int a = 5500;
   Debug.Assert(a > 10000, "Der Wert von a ist kleiner als 10000!");
```

BEISPIEL 11.8: Meldung bei Verletzung einer Annahme

Ergebnis

Nach Programmstart erscheint ein monströses Meldungsfenster, das hier nur in stark reduzierter Form dargestellt werden soll:

Fail

Diese Methode funktioniert ähnlich wie *Assert* und bietet eine elegante Alternative zum Aufruf von *MessageBox.Show*.

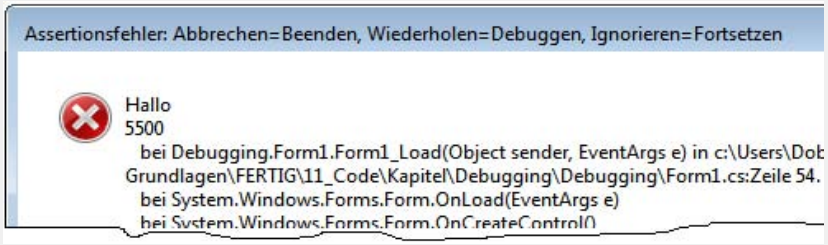
BEISPIEL 11.9:

C#

```
using System.Diagnostics;
...
   int a = 5500;
   Debug.Fail("Hallo", a.ToString());
```

Ergebnis

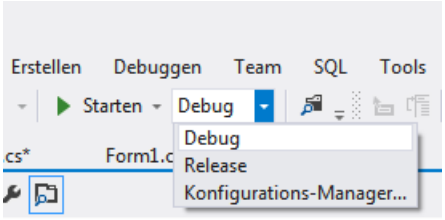
Das Meldungsfenster entspricht prinzipiell dem beim Aufruf der *Assert*-Methode (siehe obiges Beispiel).



Gegenüber *MessageBox.Show* hat die *Fail*-Methode den Vorteil, dass Sie die Auswahl haben zwischen Abbruch, Debugging oder Fortsetzung des Programms.

11.2.2 Besonderheiten der Trace-Klasse

Bislang hatten wir keinerlei Unterschiede zwischen den Klassen *Debug* und *Trace* festgestellt. Und dennoch gibt es einen, der sich allerdings erst beim Wechsel der Build-Konfiguration bemerkbar macht. Diese stellen Sie mit der kleinen Klappbox in der Symbolleiste von Visual Studio ein (standardmäßig *Debug*).



Sicher haben Sie im *bin*-Unterverzeichnis Ihres Projektordners bereits die Unterverzeichnisse *Debug* und *Release* entdeckt. Je nach gewählter Build-Konfiguration wird das Kompilat in einen dieser beiden Ordner abgelegt.

Und nun kommen wir zum "feinen" Unterschied: Grundsätzlich werden Aufrufe der Klasse *Trace* immer kompiliert, also unabhängig von der Build-Konfiguration. Wenn Sie aber *Release* wählen, werden vom Compiler nur noch Aufrufe der Klasse *Trace* verarbeitet, die der Klasse *Debug* werden ignoriert.

HINWEIS: Standardmäßig sind in der Release-Build keinerlei Debug-Informationen enthalten!

Es ist zu beachten, dass *Trace*-Aufrufe immer zu einer größeren EXE- oder DLL-Datei führen. Der Vorteil aber ist, dass auch später am fertigen Programm noch Zustände zur Laufzeit geprüft, Fehler lokalisiert und Fehlermeldungen ausgegeben werden können.

11.2.3 TraceListener-Objekte

Nicht immer ist das Ausgabefenster in Visual Studio der ideale Ort, um die Ergebnisse des Debuggings auszuwerten, denn nach Beenden der Anwendung sind sie auf Nimmerwiedersehen verschwunden. Mit Hilfe von *TraceListener*-Objekten kann man die Meldungen auch in eine Datei oder in das Windows-Ereignisprotokoll umleiten. Das .NET-Framework bietet zahlreiche vordefinierte Listener an, auf die wir aber hier aus Platzgründen nicht näher eingehen können (siehe Online-Hilfe). Lediglich die Klassen *TextWriterTraceListener* und *EventLogTraceListener* sollen – aufgrund ihrer Wichtigkeit – hier kurz beschrieben werden:

TextWriterTraceListener

Sie können diese Klasse verwenden, um um Debug-Meldungen in eine Textdatei zu schreiben. Das ermöglicht es Ihnen, Ihre Applikation auch außerhalb der Entwicklungsumgebung zu verwenden und trotzdem ein Fehlerprotokoll zu erhalten.

BEISPIEL 11.10: Debugausgaben in eine Textdatei umleiten

C#
```
using System.Diagnostics;
...
   TextWriterTraceListener myListener =
                              new TextWriterTraceListener(@"C:\debugProtocol.txt");
   Trace.Listeners.Add(myListener);
   Trace.WriteLine("Hier ist eine Debug-Meldung!");
   myListener.Flush();               // leert Ausgabepuffer und schreibt Inhalt in Datei
   myListener.Close();               // schließt den Listener
```

Die *Flush*-Methode leert den Ausgabepuffer und schreibt dessen Inhalt in die im Konstruktor angegebene Datei. Alternativ könnten Sie aber auch die *AutoFlash*-Eigenschaft auf *true* setzen.

App.config zur Definition verwenden

Einfacher und eleganter ist es, wenn Sie den Listener nicht per C#-Quellcode (siehe obiges Beispiel), sondern direkt in der *Anwendungskonfigurationsdatei* definieren. Falls eine *App.config* noch nicht existiert, können Sie sie über das Menü *Projekt/ Neues Element hinzufügen...* (oder über das entsprechende Kontextmenü im Projektmappen-Explorer) erzeugen.

Zwischen die beiden *configuration*-Tags fügen Sie dann den im Folgenden fett hervorgehobenen XML-Code ein:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="true">
      <listeners>
        <add name="debugListener"
          type="System.Diagnostics.TextWriterTraceListener"
          initializeData="debugProtocol.txt" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

HINWEIS: Je nach Build-Konfiguration wird die Protokolldatei in das *\bin\Debug*- oder in das *\bin\Release*-Unterverzeichnis Ihres Projektordners geschrieben.

Nachdem Sie obige *App.config* erstellt bzw. ergänzt haben, können Sie beliebige Ausgaben der Klassen *Debug* oder *Trace* in der Protokolldatei ablegen.

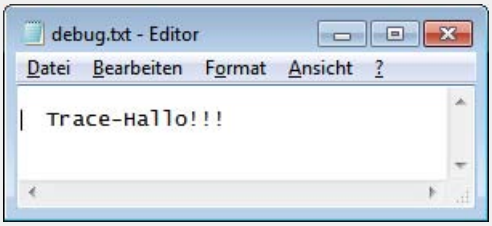
BEISPIEL 11.11: Test der obigen App.config

C#
```
using System.Diagnostics;
...
    Trace.WriteLine("Trace-Hallo!!!");
```

BEISPIEL 11.11: Test der obigen App.config

Ergebnis

Öffnen Sie die im *\bin\Debug*-Unterverzeichnis angelegte Datei:



Solange Visual Studio noch geöffnet ist, dürfte auch das Ausgabefenster (*Fenster/Ausgabe*) einen (bis auf den Vorspann) identischen Inhalt wie obige Textdatei anzeigen.

EventLogTraceListener

Wem eine einfache Textdatei zu schnöde ist, der kann die Ausgaben von *Debug*- und *Trace*-Objekt auch in die Ereignisanzeige von Windows schreiben lassen.

BEISPIEL 11.12: *Trace*-Ausgaben in das Ereignisprotokoll schreiben

C#

Das gewünschte Protokoll aussuchen:

```
EventLog log = new EventLog("Application");
```

Anwendungsname hinzufügen:

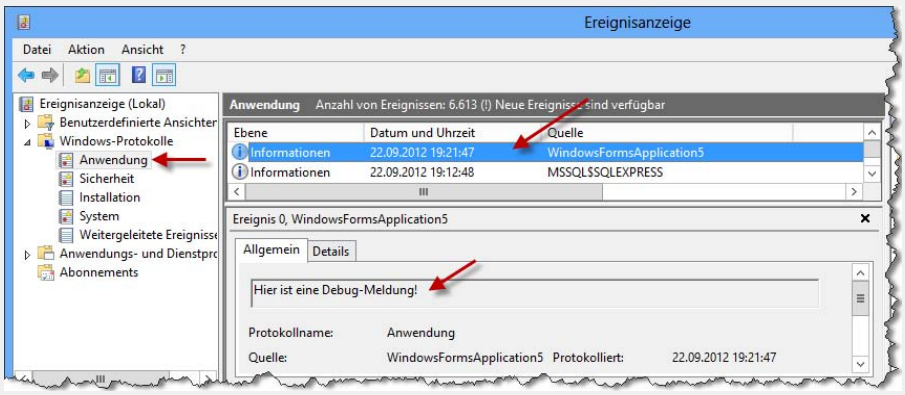
```
log.Source = Assembly.GetEntryAssembly().GetName().Name;
```

EventLogTraceListener erzeugen und nutzen:

```
EventLogTraceListener myListener = new EventLogTraceListener(log);
Trace.Listeners.Add(myListener);
Trace.WriteLine("Hier ist eine Debug-Meldung!");
myListener.Close();
```

...

Ergebnis



11.3 Caller Information

Dieses erst mit C# 5.0 eingeführte Sprachfeature unterstützt Sie bei der Fehlersuche und beim Entwickeln von Diagnose-Tools. So kann z.B. doppelter Code vermieden werden, wenn er in vielen Methoden für den gleichen Zweck aufgerufen wird, wie z.B. beim Logging und Tracing.

11.3.1 Attribute

Folgende Informationen können über Attribute der aufgerufenen Methode von der aufrufenden Methode (Caller) gewonnen werden:

- **CallerFilePath**
  der komplette Pfad der Quelldatei ,die den Aufrufer enthält (Dateipfad zur Kompilierzeit).
- **CallerLineNumber**
  Zeilennummer in der Quelldatei, in welcher die Methode aufgerufen wird.
- **CallerMemberName**
  Name des Aufrufers (Methode oder Property)

11.3.2 Anwendung

Ein einfaches Konsolenprogramm soll das Prinzip verdeutlichen:

BEISPIEL 11.13: Abfragen der Caller Information

C#

```
...
using System.Runtime.CompilerServices;
...
namespace CallerInformationNeu
{
    class Program
    {
```

Die aufrufende Methode (Caller):

```
        static void Main(string[] args)
        {
            MethodeA();              // Aufruf
            Console.ReadLine();
        }
```

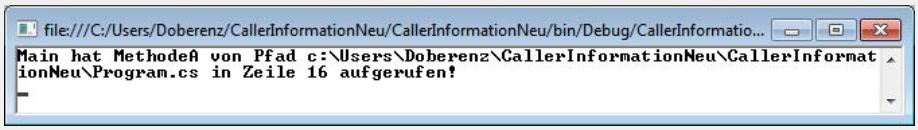
Die aufgerufene Methode:

```
        static void MethodeA([CallerMemberName] string callerName = "",
                             [CallerFilePath] string callerFilePath = "",
                             [CallerLineNumber] int callerLineNumber = 0)
        {
```

BEISPIEL 11.13: Abfragen der Caller Information

```
            Console.WriteLine("{0} hat MethodeA von Pfad {1} in Zeile {2} aufgerufen!",
                              callerName, callerFilePath,
                              callerLineNumber.ToString());
        }
    }
}
```

Ergebnis



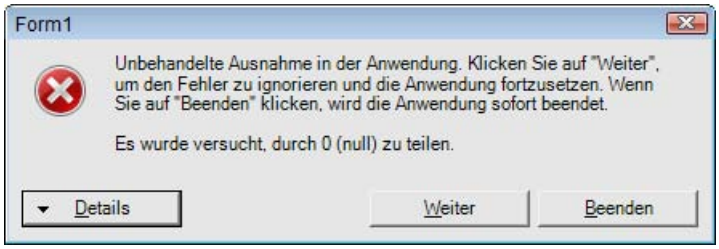
11.4 Fehlerbehandlung

Neben den logischen Fehlern, die Sie mit dem Debugger aufspüren können, enthält jeder Code auch eine ganze Reihe weiterer potenzieller Fehlermöglichkeiten, auf die Sie als Programmierer gefasst sein müssen. Beispiele für solche Fehler gibt es viele:

- falsche bzw. unerwartete Anwendereingaben in einem Formular
- eine fehlerhafte Anmeldung am SQL Server
- gesperrte Dateien, fehlende Disketten etc.
- zu wenig Speicher
- die gefürchtete Division durch null und andere mathematische Unwägbarkeiten

11.4.1 Anweisungen zur Fehlerbehandlung

Ist Ihr Programm auf einen Fehler nicht vorbereitet, wird der bedauernswerte Anwender zum Beispiel mit folgender Meldung konfrontiert:



Sicherlich wollen Sie einen solchen Anblick nicht zum Markenzeichen Ihrer Programme werden lassen. Das haben Sie auch nicht nötig, denn C# bietet genügend Möglichkeiten für eine angemessene Fehlerbehandlung.

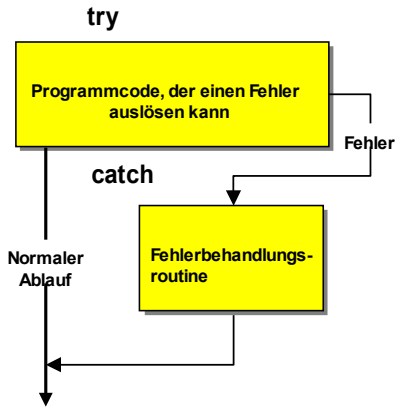
Drei Konstrukte zur Fehlerbehandlung werden unterstützt:

- *try-catch*-Blöcke
- *try-finally*-Blöcke
- das *Application.OnThreadException*-Ereignis

11.4.2 try-catch

Programmblöcke, die einen Fehler auslösen können, werden in so genannten *try-catch*-Blöcken "gekapselt" (siehe folgende Abbildung).

Tritt innerhalb des geschützten Blocks (d h. zwischen *try* und *catch*) ein Fehler auf, wird die Programmausführung in diesem Block unterbrochen und hinter *catch* fortgesetzt. Sollte wider Erwarten doch kein Fehler auftreten, wird der *catch*-Block **nie** durchlaufen.



BEISPIEL 11.14: *try-catch*

C#
```
private void button1_Click(object sender, EventArgs e)
    {
      int a,b,c;
      a = 10;
      b = 0;
      try
      {
        c = a / b;
        this.Text = c.ToString();
      }
      catch
      {
        this.Text = "Fehler";
      }
    }
```

Tritt im *try -catch*-Block ein Fehler auf, werden alle Anweisungen im *catch*-Block ausgeführt, d h., es wird hier statt des Ergebnisses der String "Fehler" in der Titelleiste des Formulars angezeigt.

Ausnahmen über Fehlerklassen auswerten

Neben der pauschalen Anzeige einer Dialogbox können Sie auch gezielt einzelne Fehlerklassen behandeln. Verwenden Sie dazu das folgende Konstrukt.

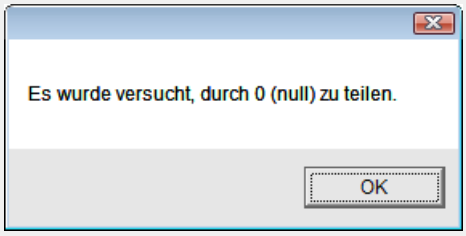
BEISPIEL 11.15: "Gezielte" Fehlerbehandlung

C#

```
int a,b,c;
a = 10;
b = 0;
try
{
  c = a / b;
  this.Text = c.ToString();
}
catch(Exception mye)
{
  MessageBox.Show(mye.Message);
  this.Text = "Fehler";
}
```

Ergebnis

In diesem Fall wird der Fehler über ein *Exception*-Objekt ausgewertet, Sie können sich die systeminterne Fehlermeldung anzeigen lassen:



Spezifische Fehlerklassen auswerten

Tritt ein Fehler in einem größeren Codeblock auf, ist es sicher interessant festzustellen, um welche Art von Fehler es sich handelt. Bei den bisher vorgestellten Varianten wurde Ihnen zwar der Fehlertext übergeben, dessen Auswertung dürfte jedoch viel zu aufwändig sein. Besser ist die Verwendung von spezifischen Fehlerklassen, mit denen Sie Fehler in bestimmte Kategorien einordnen und getrennt behandeln können.

BEISPIEL 11.16: Spezifische Fehlerklasse verwenden

C#

```
int a,b,c;
a = 10;
b = 0;
```

BEISPIEL 11.16: Spezifische Fehlerklasse verwenden

```
try
{
  c = a / b;
  this.Text = c.ToString();
}
```

Spezielle Fehlerklasse:

```
catch(DivideByZeroException mye)
{
  MessageBox.Show("Mal wieder eine Division durch null!");
}
```

Zur Sicherheit auch noch eine allgemeine Fehlerbehandlung:

```
catch(Exception mye)
{
  MessageBox.Show(mye.Message);
  this.Text = "Fehler";
}
```

Ein paar Probleme dieser Art von Fehlerbehandlung werden Sie sicher auch schon erkannt haben:

- Woher soll man die verschiedenen Fehlerklassen kennen?
- Welche Fehlerklasse tritt bei welcher Ausnahme auf?

Für die erste Frage dürfte es zumindest eine Antwort geben, wenn Sie unter dem Menüpunkt *Debuggen|Ausnahmen* nachsehen.

Ein weiterer Aspekt ist der doch recht hohe Code-Aufwand, wenn mehrere Fehlerarten ausgewertet werden sollen.

Fehler erneut auslösen

Bei den beiden oben genannten Varianten der Fehlerbehandlung wird die Programmausführung normal fortgesetzt. Möchten Sie jedoch, dass der Fehler zu einem Programmabbruch führt, können Sie diesen im *catch*-Block erneut auslösen. Verwenden Sie dazu die *throw*-Anweisung.

BEISPIEL 11.17: Fehler erneut auslösen

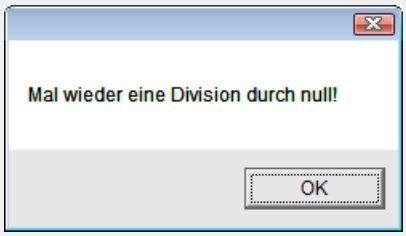
C#

```
int a,b,c;
a = 10;
b = 0;

try
{
  c = a / b;
  this.Text = c.ToString();
}
```

BEISPIEL 11.17: Fehler erneut auslösen

```
    catch(DivideByZeroException mye)
    {
      MessageBox.Show("Mal wieder eine Division durch null!");
      throw mye;
    }
```

Ergebnis



Ihre eigene Fehlermeldung wird nun zwar angezeigt, nach dem Aufruf von *throw* verhält sich das Programm jedoch so, als ob keinerlei Fehlerbehandlung vorhanden ist, es sei denn, Sie haben eine übergeordnete Fehlerbehandlung realisiert.

BEISPIEL 11.18: Weiterreichen eines Fehlers an eine übergeordnete Fehlerbehandlung

C#

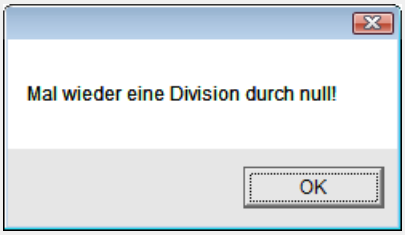
```
private void Test()
{
   int a,b,c;
   a = 10;
   b = 0;
   try  {
     c = a / b;
     this.Text = c.ToString();
   }
   catch(DivideByZeroException mye)
   {
     MessageBox.Show("Mal wieder eine Division durch null!");
     throw mye;
   }
   MessageBox.Show("Alles OK in Test!");
}

private void button5_Click(object sender, EventArgs e)
{
  try
  {
    Test();
  }
  catch
  {
    MessageBox.Show("Fehler in Test!");
```

BEISPIEL 11.18: Weiterreichen eines Fehlers an eine übergeordnete Fehlerbehandlung

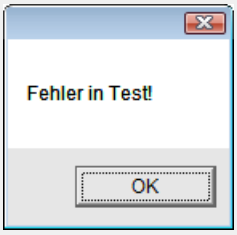
```
        }
        MessageBox.Show("Alle Ok im Hauptprogramm!");
    }
```

Was läuft hier ab? Zunächst wird durch den Button-Klick die Prozedur Test in einem *try-catch*-Block gestartet. In der Prozedur Test tritt beim Ausführen der Division ein Overflow auf, der zwar kurzzeitig abgefangen wird, aber durch die *throw*-Anweisung erneut ausgelöst wird.

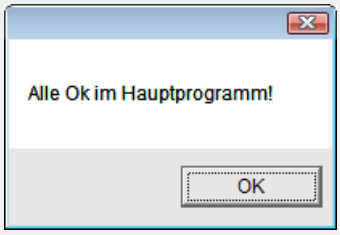
Ergebnis



Die Anweisung *MessageBox.Show("Alles OK in Test!")* wird nie ausgeführt, da der erneut auftretende Fehler direkt in die übergeordnete Fehlerbehandlung, d.h. bei *Button_Click*, verzweigt. Hier wird der Fehler endgültig abgearbeitet:

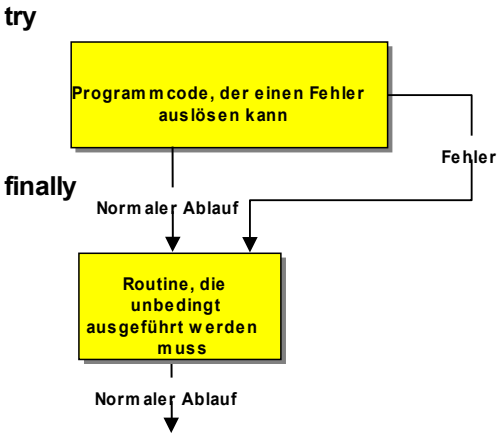


Die Programmausführung wird anschließend normal fortgesetzt:



11.4.3 try-finally

Ein weiteres Konstrukt zur Fehlerbehandlung ist dann von Interesse, wenn das "Kind schon in den Brunnen gefallen ist".



Nach dem Motto "Retten, was zu retten ist" geht es darum, Systemressourcen etc. auch im Fehlerfall sicher wieder freizugeben. Die Rede ist von den *try-finally*-Blöcken.

Sollte ein Fehler den gewohnten Ablauf stören, wird die Programmausführung im geschützten Block unterbrochen und **nach** *finally* in jedem Fall fortgesetzt.

HINWEIS: Dieser Teil wird auch beim "normalen" Programmablauf ausgeführt.

Diese Art der Fehlerbehandlung sollten Sie für das Reservieren/Freigeben von Ressourcen verwenden (Speicher, Objekte und dergleichen).

BEISPIEL 11.19: try-finally beim Versuch, zwei Access-Datenbanken zu öffnen

C#
```
using System.Data.OleDb;
...
private void button6_Click(object sender, EventArgs e)
{
  OleDbConnection conn1 = new
     OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=1.mdb;");
  OleDbConnection conn2 = new
    OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=2.mdb;");
  try
  {
    conn1.Open();
    conn2.Open();
  }
  finally
```

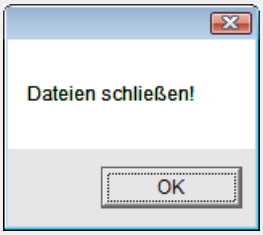
BEISPIEL 11.19: try-finally beim Versuch, zwei Access-Datenbanken zu öffnen

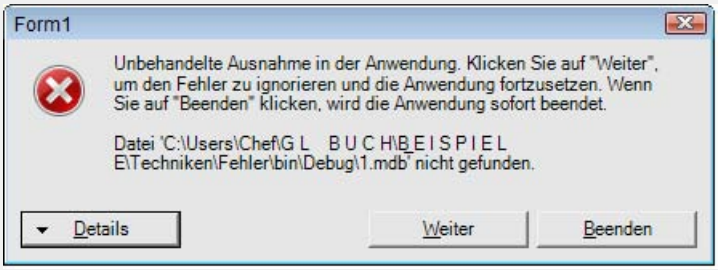
```
  {
    MessageBox.Show("Dateien schließen!");
    conn1.Close();
    conn2.Close();
  }
  MessageBox.Show("Alle Ok!");
}
```

Der Versuch, eine der beiden Connections zu öffnen, wird fehlschlagen, da die Datei nicht vorhanden ist.

Ergebnis

Nachfolgend werden folgende Dialogboxen angezeigt[1]:





Ob es zu einem Programmabbruch kommt, hängt von der Reaktion des Nutzers in der zweiten Dialogbox ab.

HINWEIS: Sie können diese Art von Fehlerbehandlung auch mit der *try-catch*-Fehlerbehandlung verbinden, um die Fehlerdialogbox zu vermeiden.

BEISPIEL 11.20: *try-catch-finally* (siehe Vorgängerbeispiel)

C#

```
OleDbConnection conn1 = new
    OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=1.mdb;");
OleDbConnection conn2 = new
    OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=2.mdb;");
```

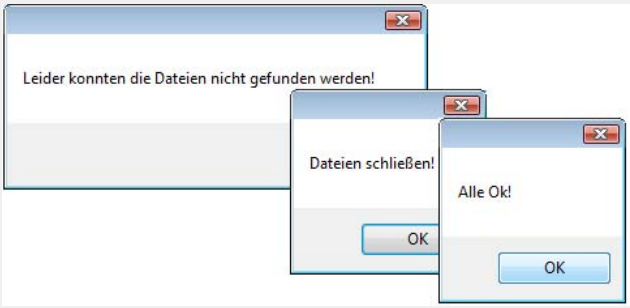
[1] In der Visual Studio-IDE erscheinen die Dialoge in einer anderen Reihenfolge, für uns ist jedoch das reine Laufzeitverhalten von Interesse.

BEISPIEL 11.20: *try-catch-finally* (siehe Vorgängerbeispiel)

```
try
  {
    conn1.Open();
    conn2.Open();
  }
  catch
  {
    MessageBox.Show("Leider konnten die Dateien nicht gefunden werden!");
  }
  finally
  {
    MessageBox.Show("Dateien schließen!");
    conn1.Close();
    conn2.Close();
  }
  MessageBox.Show("Alle Ok!");
```

Ergebnis

Die Reihenfolge der Dialogboxen:



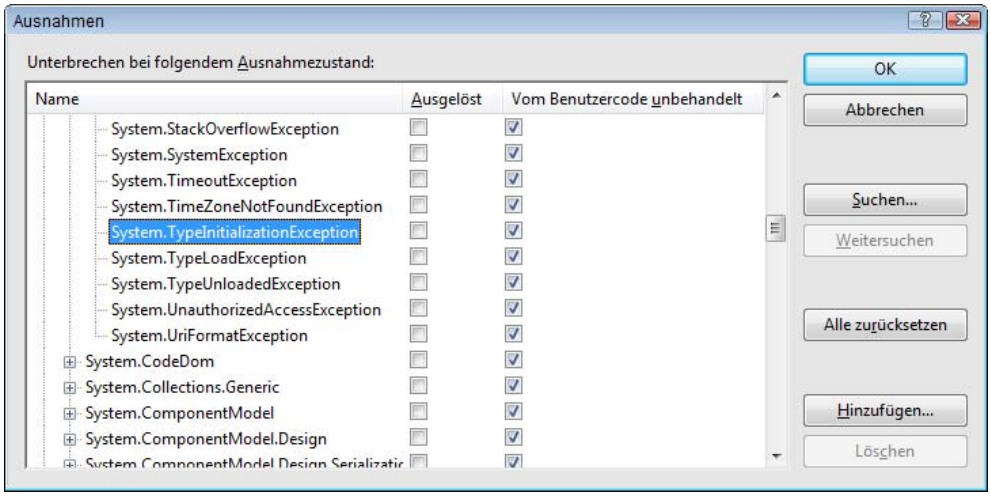
Wie Sie sehen, wird die Programmausführung anschließend normal fortgesetzt.

11.4.4 Das Standardverhalten bei Ausnahmen festlegen

Ganz kurz hatten wir ihn schon angesprochen, den Menüpunkt *Debuggen|Ausnahmen ...*, hinter dem sich u.a. folgende Dialogbox versteckt (siehe folgende Abbildung).

Neben der reinen Darstellung einer hierarchisch gegliederten Liste der möglichen Ausnahmen bietet die Dialogbox auch die Möglichkeit, das Verhalten des Debuggers bei Ausnahmen anzupassen. Zwei Optionen sind zu unterscheiden:

- das Verhalten, wenn eine Ausnahme ausgelöst wird (also noch vor der eigentlichen Fehlerbehandlung)
- das Verhalten, wenn eine Ausnahme nicht durch das Programm behandelt wird (z.B. *try-catch*)



HINWEIS: Alle hier vorgenommenen Einstellungen gelten nur für den Entwurf. Die Release-Version wird davon nicht beeinflusst.

11.4.5 Die Exception-Klasse

Was genau ist eigentlich eine Exception? In .NET ist die Antwort schnell gefunden, wie immer handelt es sich um eine Klasse bzw. ein Objekt, das über seine Eigenschaften Informationen zur gerade aufgetretenen Ausnahme bzw. zum Fehler zur Verfügung stellt:

Eigenschaft	Beschreibung
HelpLink	… Verweis (z.B. URL, Link) auf eine ausführliche Beschreibung des Fehlers (z.B. Hilfedatei)
HResult	… eine eindeutige Fehlernummer
InnerException	… ein Verweis auf die ursprüngliche Exception
Message	… ein beschreibender Text
Source	… Anwendung bzw. Objekt, in dem der Fehler aufgetreten ist
StackTrace	… liefert Informationen vom Aufruf-Stack
TargetSite	… ein Verweis auf die Methode, die die Ausnahme verursacht hat

Wie Sie eine Ausnahme auslösen bzw. wie Sie neue Ausnahmen erzeugen, zeigen Ihnen die folgenden Abschnitte. Ausgangspunkt für eigene *Exceptions* ist in jedem Fall die *Exception*-Klasse mit den o.g. Eigenschaften.

11.4.6 Fehler/Ausnahmen auslösen

Neben den vom System bzw. vom Framework ausgelösten *Exceptions* können Sie auch selbst Ausnahmen auslösen. Die dafür notwendige Anweisung *throw* hatten Sie bereits in einem der vorhergehenden Abschnitte kennen gelernt. Ziel in den vorhergehenden Abschnitten war das erneute Auslösen einer *Exception*, nachdem eine Fehlerbehandlung eingesetzt hatte. Der Fehler führte in diesem Fall entweder zur erneuten Fehlerbehandlung in einem übergeordneten *try-catch*-Konstrukt oder zu den schon bekannten Fehlerdialogboxen.

Der Anweisung selbst wird als Parameter ein initialisiertes *Exception*-Objekt übergeben. Den Typ bestimmen Sie anhand der Ausnahme.

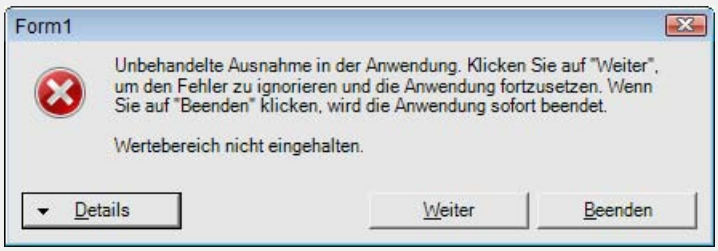
BEISPIEL 11.21: Eine Prozedur *Test* erwartet zwei Argumente *a* und *b*, die einen festgelegten Wertebereich haben. Wird dieser überschritten, löst die Prozedur eine Exception aus.

C#

```
private void test(int a, int b)
{
  if ((a>80000)|(b>100000))
      throw new System.Exception("Wertebereich nicht eingehalten");
}
```

Ergebnis

Wird die Prozedur mit ungültigen Parametern gestartet, erhalten Sie folgende Meldung:



11.4.7 Eigene Fehlerklassen

So praktikabel auch die bisherigen Varianten sind, in einigen Fällen möchte man doch eigene Fehlerklassen erzeugen, sei es, dass spezielle Parameter übergeben werden oder dass das Verhalten geändert werden soll. Auch die Auswertung der Fehler kann wesentlich differenzierter erfolgen. Last, but not least, sind eigene Fehlerklassen im Zusammenhang mit der Komponentenentwicklung interessant.

Ein vielleicht etwas abwegiges, dafür aber leicht verständliches Beispiel soll Ihnen die Vorgehensweise vermitteln.

BEISPIEL 11.22: Es soll eine neue Fehlerklasse entwickelt werden, die in der Lage ist, eine E-Mail über den aktuellen E-Mail-Client zu versenden bzw. dort zunächst anzuzeigen.

C#

Die Klasse (wir leiten von der *Exception*-Klasse ab):

```
public class clMyError : Exception
 {
```

Der Konstruktor:

```
    public clMyError(string message) : base(message)
    {
      // Konstruktor unter Angabe einer Nachricht
    }
```

Eine zusätzliche Methode zum Versenden der E-Mail:

```
    public void SendMailMessage()
    {
      System.Diagnostics.Process.Start("mailto:support@nirgendwo.de?subject" +
                                                          "=Fehler&Body=" + this.Message);
    }
}
```

Selbstverständlich könnten Sie hier auch weitere Parameter definieren, zum Beispiel eine Zieladresse etc.

Die Verwendung (Auslösen der neuen Exception):

```
    private void test(int a, int b)
    {
     if ((a>80000)|(b>100000)) throw new clMyError("Wertebereich nicht eingehalten");
    }
```

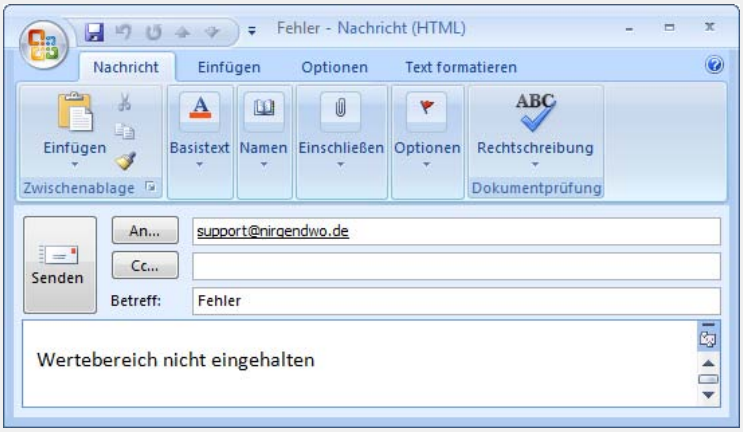
Die Verwendung (Fehlerbehandlung):

```
  private void button8_Click(object sender, EventArgs e)    {
      try
      {
        test(77777777,8);
      }
      catch  (clMyError mye)
      {
        mye.SendMailMessage();
      }
    }
```

BEISPIEL 11.22: Es soll eine neue Fehlerklasse entwickelt werden, die in der Lage ist, eine E-Mail über den aktuellen E-Mail-Client zu versenden bzw. dort zunächst anzuzeigen.

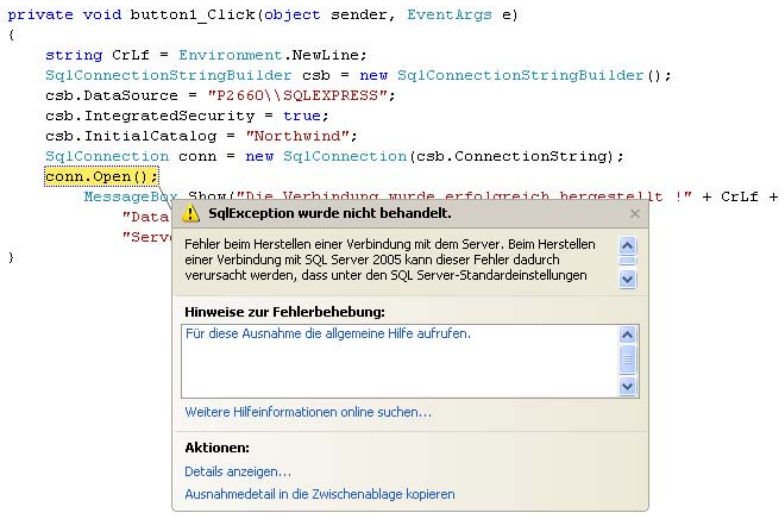
Ergebnis

Nach dem Aufruf der Methode *SendMailMessage* öffnet sich der aktuelle E-Mail-Client, die Meldungstexte und die Adresse sind bereits eingetragen.



11.4.8 Exceptionhandling zur Entwurfszeit

Auch während der Programmentwicklung erhalten Sie durch Visual Studio eine komfortable Unterstützung bei der Fehlersuche. Der Exception-Dialog ist nicht modal, d.h., Sie können mit Ihrer Arbeit am Quellcode fortfahren, ohne den Dialog vorher schließen zu müssen. Falls der Dialog stört, verschieben Sie ihn einfach in eine beliebige Ecke des Codefensters (während des Verschiebens erscheint der Dialog transparent).



Klicken Sie auf "Details anzeigen ...", um weitere Informationen zu erhalten. Außer einer Fehlerbeschreibung gibt Ihnen die Dialogbox auch Hinweise zur Fehlerbehebung mit diversen Links.

11.4.9 Code Contracts

Ab .NET 4.0 wurde die statische *System.Diagnostics.Contracts.Contract*-Klasse eingeführt. Damit können bei der Implementierung von Methoden so genannte *Code Contracts* festgelegt werden. Dabei handelt es sich um bestimmte Vereinbarungen über Parameter- und Rückgabewerte.

BEISPIEL 11.23: Eine Methode zur Division von zwei Gleitkommazahlen

C#
```
public double Dividieren(double divident, double divisor)
{
```

Der Divisor darf nicht gleich 0 sein:

```
    Contract.Assume(divisor != 0);
    return divident / divisor;
}
```

In älteren .NET-Versionen hätte man das allerdings auch mittels der *System.Diagnostics.Debug.Assert()*-Methode realisieren können. Neu ist, dass man das jetzt auch schon beim Kompilieren erkennen kann.

HINWEIS: Durch den Einsatz von Code Contracts kann man die Programmqualität spürbar verbessern, da einige Fehler nicht erst beim Testen auffallen.

Teil III

Teil III: WPF-Anwendungen

- **Einführung in WPF**
- **Übersicht WPF-Controls**
- **Wichtige WPF-Techniken**
- **Grundlagen WPF-Datenbindung**
- **Drucken/Druckvorschau in WPF**

Kapitel 12

Einführung in WPF

Wer innovative und optisch anspruchsvolle Windows Desktop-Anwendungen entwerfen will, kommt nicht mehr um die Verwendung von WPF, dem Nachfolger der Windows Forms, herum[1].

WPF ist die Abkürzung für *Windows Presentation Foundation*. Hierbei handelt es sich im weitesten Sinne um eine weitere Windows Präsentations-Schnittstelle bzw. ein Framework für die Entwicklung interaktiver Anwendungen. Eines der wesentlichsten Merkmale ist die strikte Trennung von Präsentations- und Geschäftslogik, basierend auf der neuen Beschreibungssprache XAML.

Wenn Sie bisher lediglich Windows Forms-Anwendungen entwickelt haben, sollten Sie sich zunächst die Frage stellen, welchen Typ von Anwendung Sie eigentlich jetzt entwickeln wollen:

- Windows Desktop-Anwendungen (lauffähig auf allen Windows Versionen ab Windows XP)
- Windows Store-Apps (nur Windows 8 bzw. Windows RT)

Für die Entwicklung von Windows Store-Apps müssen Sie auf WinRT setzen, eine Technologie, die in weiten Teilen auf den Grundkonzepten von WPF basiert. Mehr dazu ab Kapitel 17.

HINWEIS: Wir setzen im Teil "WinRT" weitgehende Kenntnisse von WPF-Techniken voraus, Sie können die folgenden Kapitel also auch getrost als Einführung für die WinRT-Entwicklung betrachten.

Wer den Umstieg von Windows Forms nach WPF wagt, muss mit beträchtlichem Portierungsaufwand rechnen, da vielfach gänzlich andere Konzepte zum Einsatz kommen. Eine Teilmigration Ihrer Anwendung wird von Microsoft durch das *ElementHost*-Control unterstützt, dieses kann WPF-Elemente innerhalb einer Windows Forms-Anwendung darstellen. Im Umkehrschluss lassen sich Windows Forms-Elemente per *WindowsFormsHost* auch in WPF-Anwendungen nutzen.

Bei Neuentwicklungen für den Desktop sollten Sie jedoch in jedem Fall die Verwendung von WPF in Betracht ziehen, dies auch im Hinblick auf eine Wiederverwendbarkeit Ihres Codes im Rahmen einer Webapplikation, d h. in diesem Fall als Silverlight-Anwendung.

---

[1] Aus diesem Grund haben wir auch die Windows Forms-Kapitel in das E-Book "verbannt".

Und was ist mit Silverlight?

Da die Konzepte und Technologien von WPF- und Silverlight-Anwendungen weitgehend übereinstimmen, wollen wir an dieser Stelle ebenfalls kurz darauf eingehen, aus Platzgründen musste das entsprechende Kapitel 43 jedoch in das E-Book ausgelagert werden.

12.1 Neues aus der Gerüchteküche

Leider ist die Produktpolitik von Microsoft in den letzten Jahren immer verworrener geworden. Für den Entwickler steht meist die Frage im Vordergrund, welche Technologie auch noch in einigen Jahren verfügbar ist und unterstützt wird.

Nach stürmischen Entwicklungen bei WPF und Silverlight mit teilweise rasanten Versionswechseln, hat Microsoft nun mit WinRT ein neues Pferd in den Stall gestellt. Gleichzeitig ist die Entwicklung bei Silverlight (aktuell Version 5) und WPF fast zum Erliegen gekommen.

12.1.1 Silverlight

Für Silverlight lässt sich sagen, dass es sich vermutlich um die letzte offizielle Version handeln wird. Ein Support ist seitens Microsofts jedoch bis ins Jahr 2021 gesichert. Es spricht also nichts dagegen, auch heute noch Projekte oder Teilprojekte in Silverlight zu realisieren, zumal die potenzielle Installationsbasis (ab Windows XP) auch auf Jahre hinaus noch wesentlich größer sein wird als zum Beispiel für eine reine WinRT-App, die nur auf Windows 8-Systemen lauffähig ist. Ein Blick in die Praxis zeigt recht schnell, dass auch heute noch viele Unternehmen auf Windows XP setzen bzw. derzeit am Umstieg auf Windows 7 arbeiten. Wer hier Intranet-Anwendungen realisieren will, sollte auf jeden Fall einen Blick auf Silverlight werfen.

Etwas anders sieht es mit der Unterstützung für die Entwicklung von Windows Phone-Apps aus. Hier hält Microsoft mit dem Umstieg auf die Version 8 eine Überraschung bereit, die neue Basis ist WinRT und nicht mehr XNA bzw. Silverlight.

12.1.2 WPF

Auch dem WPF-Programmierer wird nicht verborgen geblieben sein, dass sich die Neuerungen in der aktuellen Version in recht engen Grenzen bewegen. Dies kann zum einen auf fehlende Ressourcen (WinRT-Entwicklung), zum anderen aber auch auf einen Paradigmenwechsel zurückzuführen sein. Microsoft hat ein großes Interesse daran, am Tablet-Markt und am Verkauf von WinRT-Apps über den Microsoft Store zu partizipieren. Ob dieser Wunsch auch mit den Realitäten und den Vorstellungen der Entwickler übereinstimmt, ist eine andere Frage. Ihnen muss klar sein, dass Sie auf nicht absehbare Zeit sowohl WinRT- als auch Windows-Desktop-Anwendungen parallel entwickeln müssen, wenn Sie nicht auf große Zielgruppen verzichten wollen. Eine umfangreiche Business-Anwendung ausschließlich für Windows 8 zu erstellen ist sicher keine gute Idee. Nutzen Sie also für die Desktop-Entwicklung am besten WPF und profitieren Sie von den vielen Gemeinsamkeiten mit WinRT.

12.2 Einführung

Bevor wir uns im Gestrüpp der WPF-Programmierung verirren, wollen wir zunächst einen Blick auf die Grundkonzepte, Vor- und Nachteile sowie die möglichen Anwendungstypen werfen.

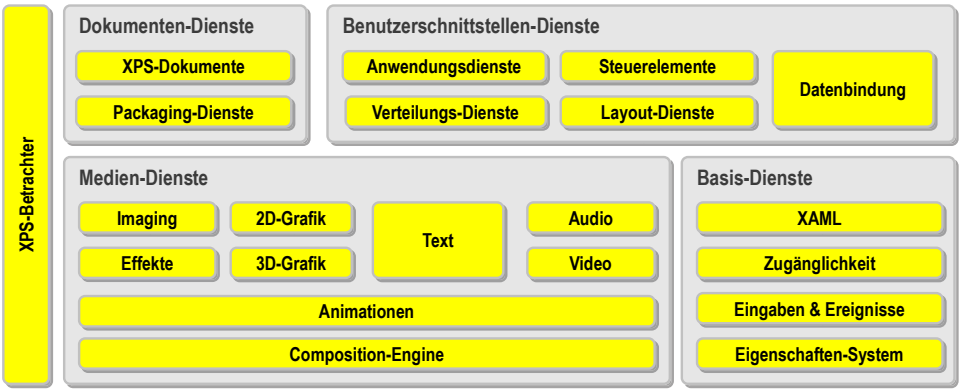
12.2.1 Was kann eine WPF-Anwendung?

Nachdem Sie die letzten 10-15 Jahre mit fast den gleichen Bibliotheken und APIs (GDI32, User32) gekämpft haben, stellt sich bei einer so radikalen Änderung zunächst die Frage, was kann WPF bzw. was macht den Unterschied zur bisherigen Vorgehensweise aus? Statt vieler Worte also zunächst eine kurze Liste der Highlights:

- Hierarchische Oberflächenbeschreibung mit XAML (XML), alternativ können die Oberflächen auch per Code beschrieben werden,
- Codebehind-Modell, ähnlich wie bei ASP.NET-Anwendungen,
- Strikte Trennung von Design und Logik,
- Desktop- (Fenster-basiert) oder Browser-Anwendung (Seiten-basiert) möglich,
- vektorbasierte Grafikausgabe (Fließkomma-Arithmetik), damit frei skalierbare Oberflächen unabhängig vom Ausgabegerät,
- schnelle Grafikausgabe dank Hardwarebeschleunigung und Ausgabe per DirectX,
- umfangreiche Unterstützung für 2D- und 3D-Grafik,
- hervorragende Layout-Optionen für Texte und Steuerelemente,
- umfangreiche Unterstützung für Grafik-Effekte (Schatten, Transparenz, Rotation, Scherung etc.) bei der Oberflächengestaltung,
- komplexe grafische Animationen für Elemente,
- Unterstützung von Medien (Videos, Bilder, Audio),
- einfache Datenbindung für fast alle Eigenschaften möglich
- ClickOnce oder XCopy-Deployment,
- teilweise Abwärtskompatibilität durch Windows Forms-Integration,
- Unterstützung für Webanwendungen in Form von Silverlight-Applikationen.

Sicherlich haben Sie in obiger Liste bereits ein Feature gefunden, das Sie bisher schmerzlich vermisst haben oder nur recht umständlich realisieren konnten.

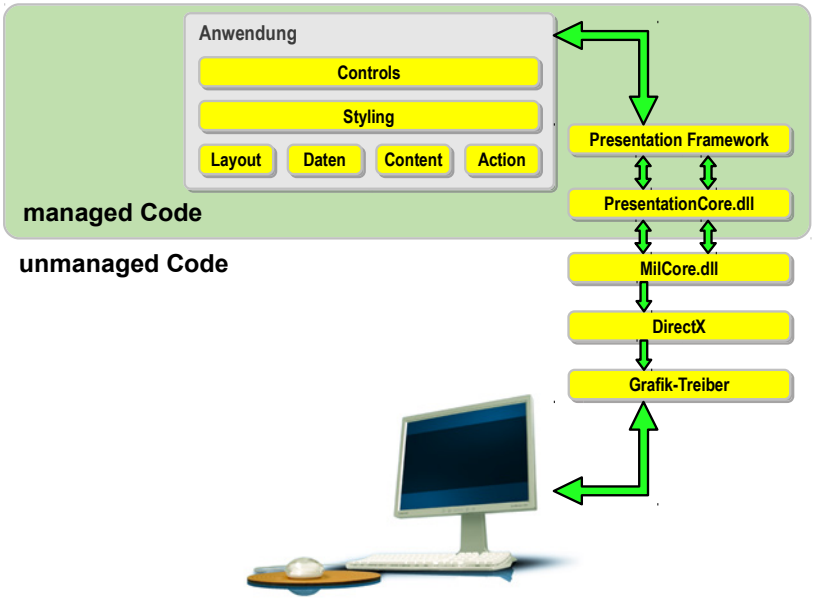
WPF soll nun all diese Möglichkeiten nicht nur unter einem Dach vereinigen, sondern die einzelnen Konzepte auch sinnvoll miteinander verzahnen. Microsoft teilt die einzelnen Funktionen dazu in eine Reihe von Diensten auf, die Sie als Entwickler in Anspruch nehmen können. Die folgende Abbildung zeigt eine entsprechende Übersicht:



Alle obigen Dienste liegen als managed Code vor, verwenden das .NET-Framework und setzen auf DirectX für die Grafikausgabe auf.

HINWEIS: Auf die Besonderheiten von *Silverlight* gehen wir erst im Rahmen des Kapitels 43 gesondert ein, dazu fehlt uns an dieser Stelle einfach der Platz.

Die folgende Abbildung zeigt Ihnen in einer Übersicht, welche Teile von WPF als managed Code vorliegen und welche Abhängigkeiten zwischen den einzelnen Ebenen bestehen.



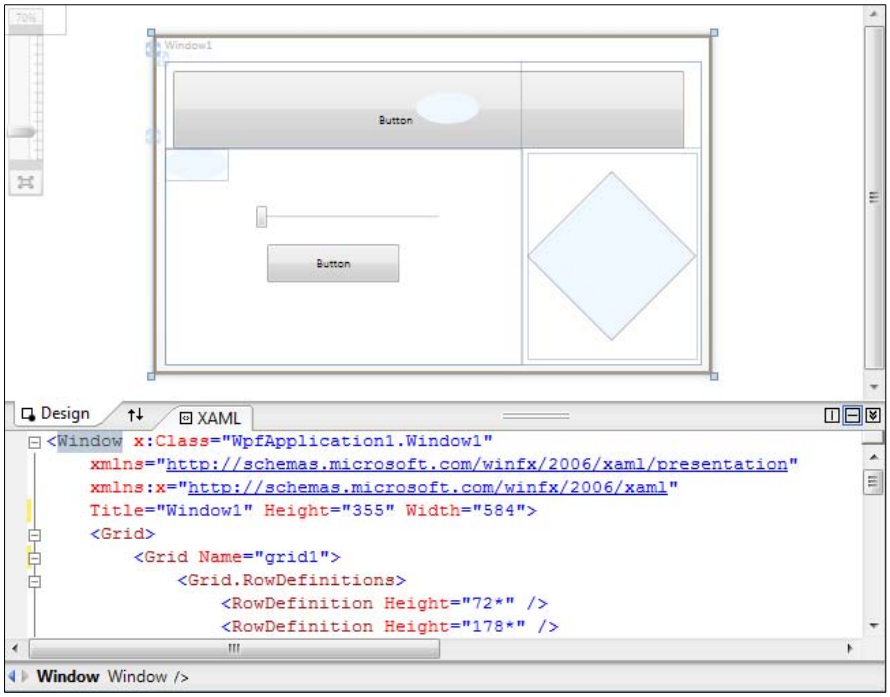
HINWEIS: Aus Performance-Gründen ist die *MilCore.dll* in unmanged Code geschrieben, hier laufen alle Grafikausgaben der Anwendung durch.

12.2.2 Die eXtensible Application Markup Language

Wie schon kurz erwähnt, basieren WPF-Anwendungen im Normalfall[1] auf einer Trennung von Oberflächenbeschreibung und Quellcode, wie Sie es auch von ASP.NET-Anwendungen her kennen. Die Oberfläche selbst wird mit einer XML-basierten Beschreibungssprache namens XAML (*eXtensible Application Markup Language* (gesprochen "Xemmel") definiert, die Programmlogik schreiben Sie wie gewohnt in C# oder VB.

Diese strikte Trennung ermöglicht es auch, die Oberfläche von einem Designer und die Logik von einem Programmierer erstellen zu lassen. Wer jetzt denkt, als Programmierer arbeitslos zu werden, braucht keine Sorge zu haben. Das grundsätzliche Programmgerüst werden Sie nach wie vor entwerfen, die optische Gestaltung (Aussehen von Controls, Animationen Grafiken etc.) kann dann ein Grafiker übernehmen.

Über grundlegende XAML-Kenntnisse sollten Sie als Programmierer ebenfalls verfügen. Auch wenn Ihnen der Visual Studio-Designer viel Arbeit abnimmt, ist es häufig wesentlich einfacher, ein paar Tags in das XAML-Dokument einzufügen, als mühevoll die Oberfläche zusammenzuklicken und per Eigenschafteneditor zu konfigurieren. Diese Tatsache scheint auch Microsoft nicht entgangen zu sein, der XAML-Code wird im Editor parallel zum Designer angezeigt, Änderungen in einem der beiden Editoren wirken sich wechselseitig aus (siehe folgende Abbildung).



[1] Sie könnten auch eine Anwendung nur per Code erzeugen, aber dann gehen Sie vermutlich auch nur zu Fuß und verwenden kein Auto.

HINWEIS: Da Microsoft zwei Zielgruppen (Designer und Entwickler) mit WPF ansprechen möchte, sind auch zwei getrennte Entwicklungsumgebungen verfügbar. Vor einer sitzen Sie vermutlich gerade (Visual Studio 2012), die zweite nennt sich *Expression Blend* und richtet sich vornehmlich an Designer. Mehr Informationen dazu finden Sie im Abschnitt 14.9.1, ab Seite 781.

Unsere erste XAML-Anwendung

Statt vieler Worte wollen wir uns zunächst mit einem einfachen XAML-Beispiel beschäftigen, bevor wir uns in den Details verlieren.

BEISPIEL 12.1: Eine erste XAML-Anwendung

XAML

Öffnen Sie einen Texteditor (*Notepad*) und tippen Sie folgenden Code ein.

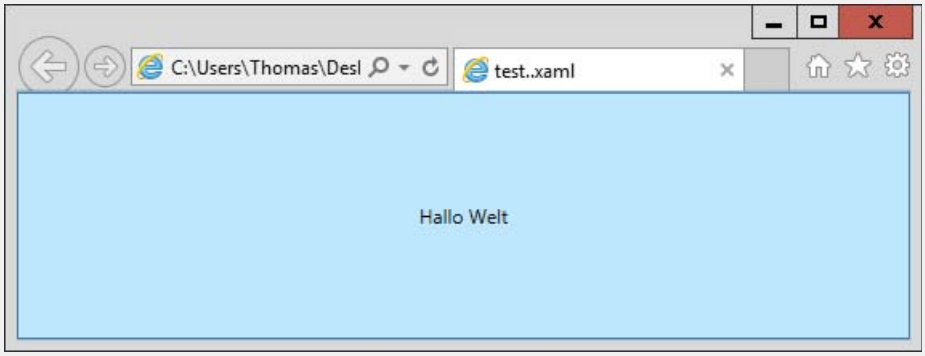
```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo Welt
</Button>
```

Speichern Sie nachfolgend die "Anwendung" unter dem Namen *Test1.xaml* ab.

HINWEIS: Die Angabe des Namespaces (*xmlns=...*) in obigem Beispiel macht dem Interpreter klar, um was für einen Button es sich eigentlich handelt. "Hallo Welt" ist der im Button enthaltenen Content, darauf kommen wir später noch zurück.

Ergebnis

Mit einem Doppelklick können wir den ersten Test starten. Ist das Framework ab Version 3 installiert, sollte im Internet-Explorer eine Schaltfläche angezeigt werden, die den gesamten Clientbereich ausfüllt:



So, das ging ja schon recht schnell, auch wenn wir noch keine Funktionalität implementiert haben. Ein paar Worte zu den Hintergründen der neuen XAML-Anwendung:

- Obige Datei wird als ungebundenes bzw. stand-alone XAML bezeichnet. Dateien mit dieser Extension sind automatisch mit dem *PresentationHost* verknüpft und werden durch diesen interpretiert.

- Da es sich um kein komplettes Programm handelt, wird automatisch ein Objekt vom Typ *Page* erzeugt, dem obiger XAML-Code als Content (Inhalt) zugewiesen wird.
- Die so erzeugten *Page*-Objekte können im Internet-Explorer angezeigt werden.

Probleme mit dem Stammelement

Mutig geworden, wollen wir nun versuchen, noch einen zweiten Button in unser Prograämmchen einzufügen.

BEISPIEL 12.2: Darstellung von zwei Schaltflächen

XAML
```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo Welt
</Button>
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
   Hallo XAML
</Button>
```

Statt der erwarteten Schaltflächen taucht im Browser eine Fehlermeldung auf, die uns mit vielen Worten den knappen Sachverhalt erklären will, dass mehr als ein Stammelement vorhanden ist.

Das scheint zwar auf den ersten Blick richtig zu sein (siehe obiges Listing), aber kurz vorher hatten wir ja erklärt, dass diese Daten als Content in ein *Page*-Objekt eingefügt werden, damit ist ja wohl ein Stammelement vorhanden. Dies ist auch korrekt, jedoch schreibt die XAML-Spezifikation vor, dass sowohl ein *Window* als auch eine *Page* nur **ein untergeordnetes Element** enthalten dürfen[1].

Dieses Regel ist für einen Windows Forms-Programmierer sicher gewöhnungsbedürftig, kann er doch dort beliebig viele Controls in ein Fenster einfügen.

Umgehen können wir diese Einschränkung, indem wir zunächst ein so genanntes Container-Control definieren, in das wir unsere Schaltflächen einfügen. WPF bietet einen reichhaltigen Fundus an Container-Klassen an, wir werden uns ab Seite 600 damit ausführlich beschäftigen.

BEISPIEL 12.3: Darstellung von zwei Schaltflächen (zweiter Versuch)

XAML
```
<StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Button>Hallo Welt</Button>
  <Button>Hallo XAML</Button>
</StackPanel>
```

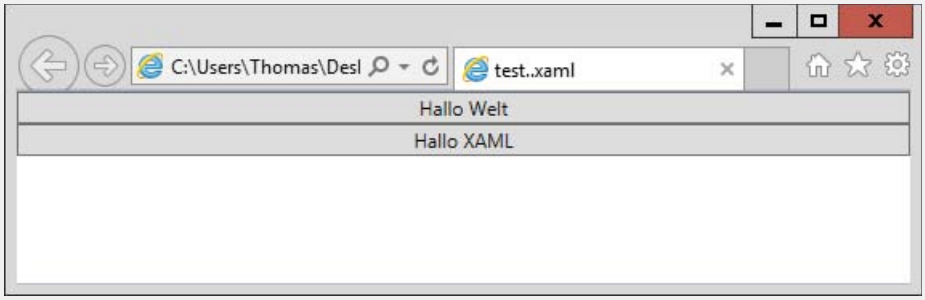
HINWEIS: Da der Namespace bereits beim *StackPanel* festgelegt wurde, können wir bei den untergeordneten Schaltflächen darauf verzichten.

[1] Darüber werden Sie sicher mehr als einmal stolpern ...

BEISPIEL 12.3: Darstellung von zwei Schaltflächen (zweiter Versuch)

Ergebnis

Ein Testlauf bringt jetzt auch die erwarteten zwei Schaltflächen auf den Bildschirm:



Aus obigem Beispiel können wir auch noch zwei weitere Erkenntnisse ableiten:

- XAML-Oberflächen sind hierarchisch aufgebaut (*Page → StackPanel → Button, Button)*
- Der Content (Inhalt) bestimmt die Abmessungen des Controls (Höhe der Schaltflächen[1])

Ein kleines Experiment

Nach so viel Erkenntnissen trauen wir uns an ein kleines Experiment. XAML-Oberflächen sind hierarchisch aufgebaut. Das wollen wir auf etwas eigenwillige Art überprüfen, indem wir einen Button in den anderen einfügen.

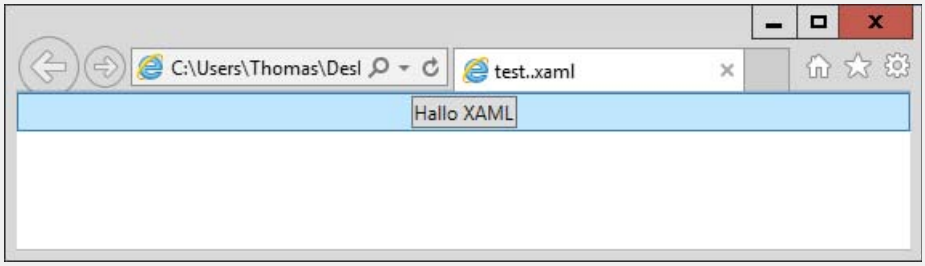
BEISPIEL 12.4: Button im Button

XAML

```
<StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" >
   <Button>
      <Button>
         Hallo XAML
      </Button>
   </Button>
</StackPanel>
```

Ergebnis

Ein Testlauf zeigt das gewünschte Ergebnis:



[1] Die Breite wird in diesem Fall durch das umgebenden *StackPanel* bestimmt.

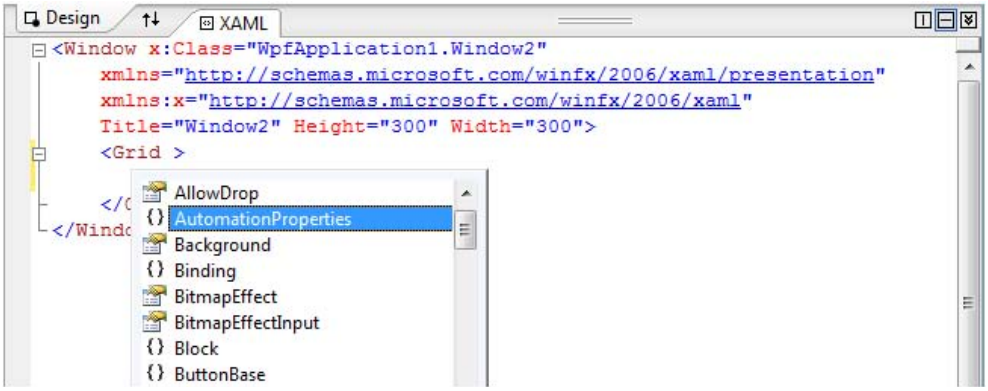
Für alle Ungläubigen: Der innere Button ist wirklich im Content des äußeren Buttons enthalten, nicht darüber. Das merken Sie schon daran, dass eine Mausbewegung über den inneren Button auch die äußere Schaltfläche aktiviert. Dies bedeutet auch, dass ein Mausklick sich auf beide Controls auswirkt (mehr dazu später).

Soll ich das alles von Hand eintippen?

Nein, natürlich nicht, wir wollten nur demonstrieren, dass Sie WPF-Oberflächen auch mit einem einfachen Editor programmieren können. Für die Eingabe stehen Ihnen in Visual Studio entweder die XAML-Ansicht oder die bekannte Design-Ansicht zur Verfügung.

Der XAML-Editor bietet mit der integrierten IntelliSense bereits eine gute Grundlage, um Controls und deren Eigenschaften (Attribute) zu definieren bzw. Ereignismethoden zuzuweisen.

Sie werden im Laufe der Zeit feststellen, dass Sie mit dem XAML-Editor recht häufig arbeiten, da dies wesentlich intuitiver als die mühsame Klickerei mit dem Designer ist.



12.2.3 Verbinden von XAML und C#-Code

Bisher haben wir uns nur mit der XAML-Oberflächenbeschreibung beschäftigt, doch so ganz ohne Code wird Ihr Programm wohl kaum auskommen. An dieser Stelle wollen wir deshalb unsere bisherigen Experimente abbrechen und uns der Projektverwaltung in Visual Studio zuwenden.

Öffnen Sie ein neues Projekt und wählen Sie den Projekttyp "WPF-Anwendung". Visual Studio erstellt daraufhin vier Dateien, zwei XAML-Dateien (*App.xaml*, *MainWindow.xaml*) und zwei dazugehörige Klassendefinitionen (*App.xaml.cs*, *MainWindow.xaml.cs*) in C#.

Schauen wir uns diese Dateien einmal im Detail an.

App.xaml

In dieser Datei wird, neben der Einbindung der beiden obligatorischen Namespaces, die Klasse *App* konfiguriert. Im vorliegenden Fall wird mit dem Attribut *StartupUri* festgelegt, welches Fenster als erstes angezeigt wird.

BEISPIEL 12.5: ***App.xaml***

XAML
```
<Application x:Class="WpfApplication1.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

HINWEIS: Der Bezug zur Hintergrundcodedatei wird über das Attribut *x:Class* hergestellt (*Namespace.<Klassennamen>*).

Ganz nebenbei nimmt diese Datei auch noch die Definition von Anwendungsressourcen (globale Ressourcen) auf, wir kommen ab Seite 739 darauf zurück.

App.xaml.cs

Hierbei handelt es sich um die Hintergrundcodedatei zu obiger XAML-Datei mit der noch leeren Klassendefinition:

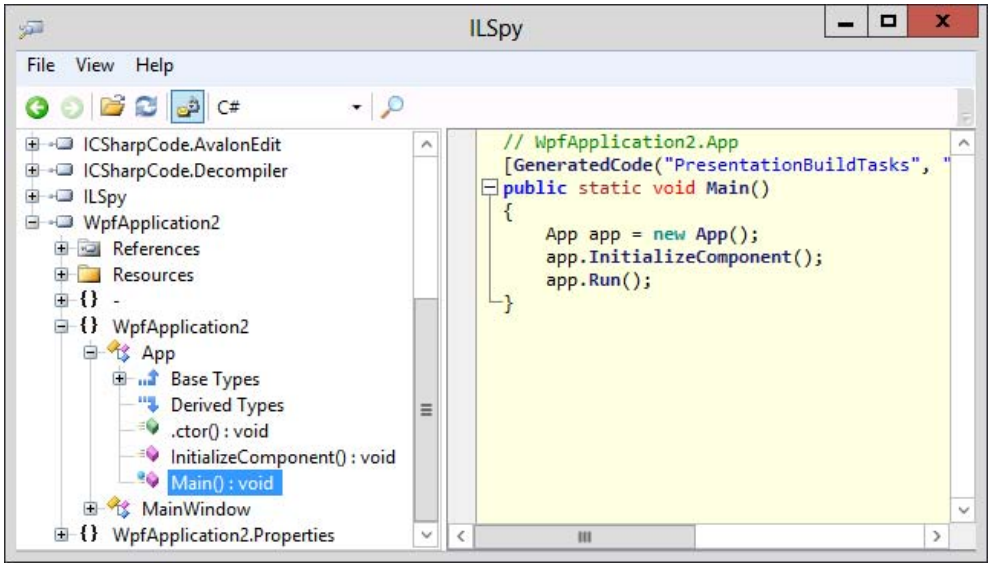
BEISPIEL 12.6: ***App.xaml.cs***

C#
```
using System;
using System.Collections.Generic;
...
using System.Windows;

namespace WpfApplication1
{
...
    public partial class App : Application
    { }
}
```

Die derart definierte *App*-Klasse können Sie zur Laufzeit über *App.Current* abrufen. Die Klasse selbst stellt bereits einen umfangreichen Satz an Ereignissen und Eigenschaften zur Verfügung (mehr dazu ab Seite 621).

Wer das eigentliche Hauptprogramm in dieser Datei vermutet hat, liegt falsch, denn dieses wird von Visual Studio automatisch erzeugt. Zusätzlich wird noch eine Methode *InitializeComponent* angelegt, in der das Startformular entsprechend der Optionen in *App.xaml* festgelegt wird.

Wer es nicht glaubt, der kann sich mit dem ILSpy davon überzeugen, dass die finale Assembly über ein Hauptprogramm verfügt (siehe folgende Abbildung).



HINWEIS: Das obige *STAThread*-Attribute gibt an, dass das COM-Threadingmodell für die Anwendung *Singlethread-Apartment* ist. Dieses Attribut muss vorhanden sein, andernfalls funktionieren die Controls/Windows nicht richtig.

BEISPIEL 12.7: Der Inhalt der Methode *InitializeComponent*

C#
```
[DebuggerNonUserCode]
public void InitializeComponent()
{
    base.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
}
```

MainWindow.xaml

Hier haben wir es mit der XAML-Beschreibung des ersten bereits automatisch erzeugten Fensters zu tun. Den Bezug zum C#-Sourcecode stellt das *x:Class*-Attribut her, die Einbindung der beiden obligaten Namespaces dürfte Ihnen bereits aus der *App.xaml* bekannt sein:

BEISPIEL 12.8: MainWindow.xaml

C#
```
<Window x:Class="WpfApplication2.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>
    </Grid>
</Window>
```

Mit dem Attribut *Title* wird der Inhalt der Kopfzeile definiert, *Width,* und *Height* dürften selbsterklärend sein.

Das *Grid*-Element definiert bereits den grundlegenden Container (und damit das Layout) für das neue Fenster, dieses Element können Sie bei Bedarf löschen und durch eines der anderen Container-Elemente ersetzen (siehe Seite 600).

MainWindow.xaml.cs

Die letzte der vier Dateien dürfte die "Spielwiese" für Sie als Programmierer sein, hier haben Sie es mit der Code-Hintergrunddatei für das mit *MainWindow.xaml* definierte Fenster zu tun.

BEISPIEL 12.9: MainWindow.xaml.cs

C#
```
using System;
...
using System.Windows.Shapes;

namespace WpfApplication2
{
...
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

In dieser Datei werden die Ereignis-Handler, zusätzliche Methoden und lokale Variablen abgelegt.

Ein erster Ereignis-Handler

Sicher wollen Sie endlich auch etwas Aktivität in unser Programm bringen, fügen Sie dazu einen einfachen *Button* in den Designer und damit in das *Grid*!!! von *MainWindow* ein. Alternativ können Sie auch die XAML-Datei von *MainWindow* bearbeiten:

BEISPIEL 12.10: Ein erster Ereignis-Handler

XAML
```
...
    <Grid>
      <Button>Ein erster Test</Button>
    </Grid>
...
```

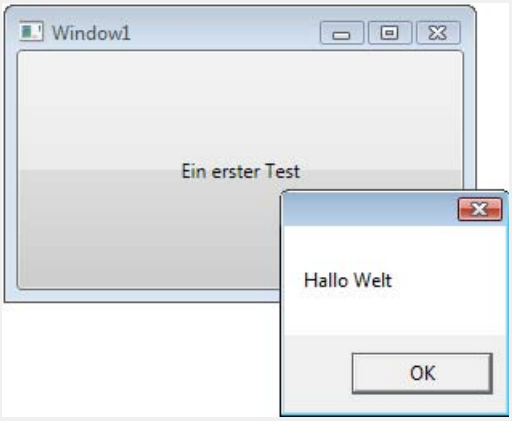
C#

Nach einem Doppelklick auf den Button erzeugt Visual Studio für Sie bereits den Ereignis-Handler. Fügen Sie jetzt noch eine *MessageBox.Show ...* ein, und Sie haben ein funktionstüchtiges Programm, bei dem XAML-Oberfläche und C#-Code miteinander verbunden sind.

BEISPIEL 12.10: Ein erster Ereignis-Handler

```
private void Button_Click(object sender, RoutedEventArgs e)
{
   MessageBox.Show("Hallo Welt");
}
```

Ergebnis



Sicher nicht ganz uninteressant ist jetzt die XAML-Datei, hier muss ja ein Verweis auf die zugehörige Ereignis-Routine eingefügt werden. Und siehe da, dem Attribut *Click* wird der Name der Ereignis-Methode übergeben:

```
...
    <Grid>
      <Button Click="Button_Click">Ein erster Test</Button>
   </Grid>
...
```

Und wo ist mein Button-Objekt?

Versuchen Sie jetzt einmal per Code auf den Button zuzugreifen, um zum Beispiel dessen Beschriftung (Content) zu ändern. Haben Sie den Button per XAML erzeugt, dürfte Ihnen dies schwerfallen, denn ein derartiges Objekt ist nicht zu finden. Ursache ist die fehlende Zuweisung des *Name*-Attributs in der XAML-Datei.

HINWEIS: Ein XAML-Element muss nur dann benannt werden, wenn Sie es auch per Code ansprechen wollen.

Erweitern Sie also die XAML-Definition des Buttons um ein *Name*-Attribut mit eindeutigem Namen:

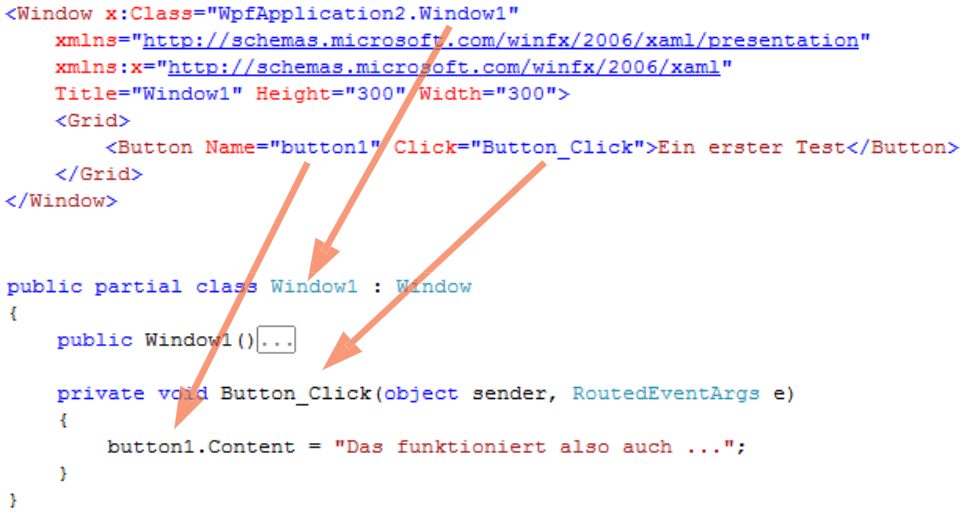
```
...
 <Grid>
    <Button Name="button1" Click="Button_Click">Ein erster Test</Button>
```

```
 </Grid>
...
```

Jetzt können Sie in der Codedatei wie gewohnt mit dem Objekt *button1* arbeiten und zum Beispiel dessen Beschriftung ändern:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    button1.Content = "Das funktioniert also auch ...";
}
```

Die folgende Abbildung zeigt noch einmal die Zusammenhänge zwischen Code und XAML:



Brauche ich unbedingt eine Trennung von Code und XAML?

Im Prinzip nein, aber eine Mischung von Code und XAML dürfte der Übersichtlichkeit Ihrer Programme sicher nicht zuträglich sein, ganz abgesehen davon, dass in diesem Fall eine getrennte Bearbeitung von Oberfläche und Logik nicht mehr sinnvoll realisierbar ist.

Aus diesen Gründen gehen wir auf die Möglichkeit, Code und XAML in einer Datei unterzubringen, nicht ein.

Kann ich Oberflächen auch per Code erzeugen?

Im Prinzip ja, der Aufwand ist jedoch teilweise recht beträchtlich, da sich die Eigenschaften und Zuweisungen leichter per XAML konfigurieren lassen.

BEISPIEL 12.11: Einen neuen *Button* in *button1* erzeugen

C#

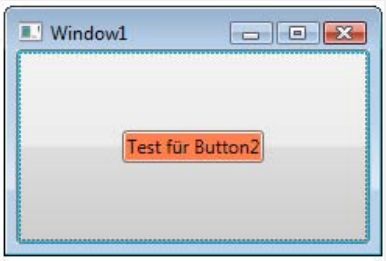
```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Button btn = new Button();
```

BEISPIEL 12.11: Einen neuen *Button* in *button1* erzeugen

```
            btn.Content = "Test für Button2";
            btn.Name = "button2";
            btn.Background = Brushes.Coral;
            button1.Content = btn;
        }
```

Ergebnis

Das Resultat zeigt die folgende Abbildung:



Dabei wollen wir es an dieser Stelle belassen, auf weitere Details werden wir im Verlauf des Kapitels zu sprechen kommen.

12.2.4 Zielplattformen

Für den Entwickler sicher nicht ganz uninteressant dürfte die Frage nach den Zielplattformen für die erstellten WPF-Anwendungen sein. Hier müssen Sie sich auf Plattformen beschränken, auf denen das .NET-Framework mindestens ab der Version 3.0 läuft.

HINWEIS: In diesem Fall müssen Sie die Zielplattform in den Projekteigenschaften explizit auf 3.0 festlegen. Möchten Sie neuere Feature in Ihrer Anwendung unterstützen, wählen Sie als Zielplattform ".NET Framework 4.5 Client" oder ".NET Framework 4.5", wie wir es auch für dieses Buch voraussetzen.

Folgende unterstützte Betriebssysteme kommen nach obiger Forderungen infrage:

- Windows 8
- Windwos Server 2012
- Windows 7
- Windows Server 2008
- Windows Vista
- Windows XP (ab SP 3) und Windows 2003 Server (hier müssen Sie das .NET-Framework erst noch installieren)

12.2.5 Applikationstypen

Microsoft bietet in Visual Studio 2012 zur Zeit vier Templates im Zusammenhang mit WPF an:

- WPF-Anwendung
- WPF-Browseranwendung, teilweise auch als Express-Anwendung bezeichnet
- WPF-Benutzersteuerelementebibliothek
- Benutzerdefinierte WPF-Steuerelementebibliothek

Im Folgenden wollen wir uns kurz mit den Unterschieden der beiden erstgenannten Anwendungstypen auseinandersetzen, um Ihnen die Entscheidung für das eine oder andere Template zu erleichtern.

WPF-Anwendung

Dieser Anwendungstyp entspricht im Wesentlichen den bisherigen Windows Forms-Anwendungen mit folgenden Möglichkeiten:

- Die Installation erfolgt per Setup (MSI) oder ClickOnce auf dem Zielcomputer.
- Ein Verweis kann im Start-Menü bzw. in der Systemsteuerung unter *Software* erfolgen.
- Die Anwendung läuft per Default im Full Trust-Mode, d h., Sie können auf die Registry, das Dateisystem, WCF etc. zugreifen.
- Die Anwendung läuft in einem eigenen Fenster (Windows).
- Die Anwendung läuft offline.
- Updates müssen explizit installiert werden (per ClickOnce automatisierbar).

WPF-Browseranwendungen

Auch wenn sie so heißt, handelt es sich doch um keine Web-Anwendung, der Name bezieht sich lediglich auf den Anzeigeort der Anwendung.

- Diese Anwendungen laufen nur im Internet Browser mit wizardartiger Oberfläche (Navigation zwischen Pages[1]).
- Die Anwendung läuft mit stark eingeschränkten Rechten in einer Sandbox.
- Die Anwendung wird **nicht** auf dem Zielcomputer installiert.
- Ein Verweis im Start-Menü bzw. in der Systemsteuerung unter Software ist **nicht** möglich.
- Die Anwendungen werden automatisch per ClickOnce verteilt.
- Obwohl der überwiegende Teil der WPF-Features genutzt werden kann, gibt es einige Einschränkungen bei der Darstellung.

[1] Die freie Navigation durch den Anwender kann den Entwickler schnell in den Wahnsinn treiben.

12.2.6 Vor- und Nachteile von WPF-Anwendungen

Über die Vorteile von WPF-Anwendungen wissen Sie ja bereits eine ganze Menge (siehe Abschnitt 12.2.1). Der wichtigste Vorteil dürfte für die meisten sicher die phantastische Oberflächengestaltung sein. Hier handelt es sich um einen echten Quantensprung gegenüber den teilweise recht tristen Windows Forms-Anwendungen.

Doch wo viel Licht ist, da ist auch Schatten, und so werden Sie früher oder später auch einige "Haare in der Suppe" finden:

- WPF-Anwendungen sind nicht auf älteren Betriebssystemen (Windows 2000 etc.) lauffähig, was jedoch mittlerweile vernachlässigbar ist.
- Komplexere Anwendungen erfordern schnell umfangreiche Kenntnisse (steile Lernkurve).
- Nicht alle Windows Forms-Steuerelemente sind vorhanden, einige Standarddialoge fallen schnell durch ihr "altbackenes" Äußeres auf.
- Viele Konzepte in WPF richten sich an den Designer und nicht an den Entwickler.
- Das Setzen von Eigenschaften per Code ist teilweise recht aufwändig.
- Trennung von Code und Oberfläche kann Probleme beim dynamischen Erstellen von Oberflächenelemente bereiten.
- WPF-Anwendungen werden schnell zum "Selbstbedienungsladen", sowohl der C#-Code als auch die XAML-Oberflächenbeschreibung können relativ einfach aus den Anwendungen extrahiert werden (Decompiler).
- Last, but not least, ist nicht jeder Anwender davon begeistert, wenn er eine zwar bunte aber kaum funktionale Anwendung vorgesetzt bekommt. WPF "erleichtert" es dem Entwickler, konfuse und wenig intuitive Oberflächen zu gestalten, aber das haben Sie ja selbst in der Hand.

Tja, wann also sollten Sie WPF verwenden und die Lektüre dieses Kapitels fortsetzen? Immer dann, wenn Sie obige Einschränkungen nicht stören, Sie sowieso ein neues Projekt beginnen und, das sollte wohl der Hauptgrund sein, Sie einen echten Nutzen mit WPF erzielen. Ansonsten schadet es sicher nicht, wenn Sie noch eine Versionsnummer abwarten oder eventuell zu den neuen Windows Store-Apps wechseln.

12.2.7 Weitere Dateien im Überblick

Sicher sind Ihnen bei Nachforschungen in den Projektverzeichnissen auch einige Dateien aufgefallen, mit denen Sie auf die Schnelle nichts anfangen können. Wir wollen uns mit *.baml* und *.g.cs* zwei der wichtigsten Typen herausgreifen und näher anschauen.

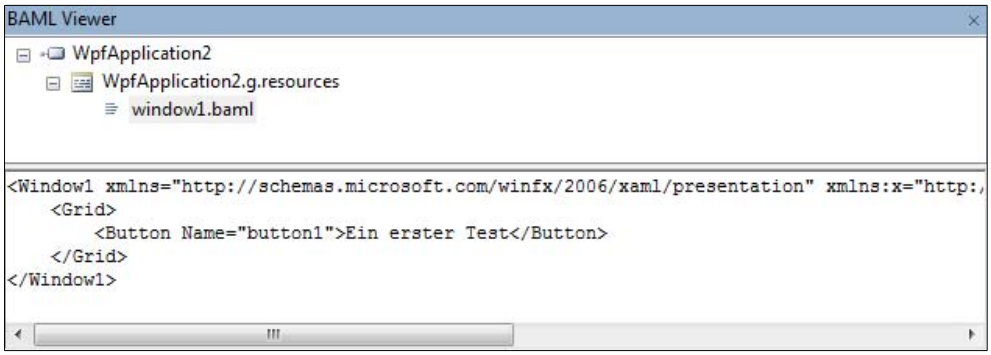
Was sind .BAML-Dateien und was passiert damit?

Bei den im Verzeichnis *\obj\...\Debug* enthaltenen Dateien handelt es sich um binäre Repräsentationen (*Binary Application Markup Language*) der einzelnen XAML-Dateien (Windows/Pages).

Diese sind nicht für die Betrachtung oder Bearbeitung vorgesehen, sondern werden als Ressourcen in die finale Assembly gelinkt.

.BAML-Dateien liegen bereits in einem Zwischenformat vor, was zur Laufzeit eine schnellere Verarbeitung des Objekt-Graphen ermöglicht.

HINWEIS: Mit einem extra Add-In ist der .NET-Reflector in der Lage, diese Ressourcen sichtbar zu machen und als lesbaren XAML-Code anzuzeigen:



Den BAML-Viewer können Sie unter folgender Adresse herunterladen:

LINK: http://reflectoraddins.codeplex.com/releases/view/1805

Worum handelt es sich bei den .G.CS-Dateien?

Das ".G" steht zunächst für "Generated". Im Detail handelt es sich um die Klassendefinition in der jeweiligen Projekt-Programmiersprache, die automatisch zu jedem XAML-File (Window/Page) angelegt wird. Diese Datei stellt den genauen Zusammenhang zwischen XAML-Elementen (Controls) und den jeweiligen C#-Klassen her.

BEISPIEL 12.12: Eine .G.CS-Datei für Windows1.xaml

C#

```
#pragma checksum "..\..\Window1.xaml" "{406ea660-64cf-4c82-b6f0-42d48172a799}"
"871E64C0DAEA9297F82991CACE5FBBA3"
...
using System;
...
using System.Windows.Shapes;
```

Der definierte Namespace:

```
namespace WpfApplication2 {
```

Die bereits in XAML definierte Klasse:

```
    public partial class Window1 : System.Windows.Window,
                                   System.Windows.Markup.IComponentConnector {
```

BEISPIEL 12.12: Eine .G.CS-Datei für Windows1.xaml

Haben Sie in die XAML-Datei benannte Controls eingefügt (*Name*-Attribut), finden Sie hier den C#-Pendant:

```
internal System.Windows.Controls.Button button1;

private bool _contentLoaded;
```

Der eigentliche Initialisierungsvorgang:

```
[System.Diagnostics.DebuggerNonUserCodeAttribute()]
public void InitializeComponent() {
    if (_contentLoaded) {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater = new
        System.Uri("/WpfApplication2;component/window1.xaml",
                   System.UriKind.Relative);
    #line 1 "..\..\Window1.xaml"
    System.Windows.Application.LoadComponent(this, resourceLocater);
}
```

Mit der folgenden Methode werden die Objekte und die Ereignisse zugeordnet:

```
[System.Diagnostics.DebuggerNonUserCodeAttribute()]
[System.ComponentModel.EditorBrowsableAttribute(
   System.ComponentModel.EditorBrowsableState.Never)]
[System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Design",
    "CA1033:InterfaceMethodsShouldBeCallableByChildTypes")]
void System.Windows.Markup.IComponentConnector.Connect(int connectionId,
        object target)
{
    switch (connectionId)
    {
    case 1:
    this.button1 = ((System.Windows.Controls.Button)(target));

    #line 6 "..\..\Window1.xaml"
    this.button1.Click += new System.Windows.RoutedEventHandler(this.Button_Click);

    #line default
    #line hidden
    return;
    }
    this._contentLoaded = true;
}
}
}
```

12.3 Alles beginnt mit dem Layout

Vielleicht erscheint Ihnen diese Überschrift etwas merkwürdig, aber nachdem Sie sich über den Typ der WPF-Anwendung (siehe Seite 596) im Klaren sind, sollten Sie sich in Erinnerung rufen, dass sowohl eine *Page* als auch ein *Window* nur über **ein** untergeordnetes Stammelement verfügen dürfen – ganz im Gegensatz zu einer Windows Forms-Anwendung, bei der Sie recht unbekümmert Controls in den Formularen verteilen können.

Aus diesem Grund werden wir uns in diesem Abschnitt zunächst mit einigen recht speziellen Controls, den Panel-Elementen oder auch Containern beschäftigen, bevor wir auf die eigentlichen Elemente der Bedienoberfläche zu sprechen kommen (Schaltflächen, Listenfelder etc.).

12.3.1 Allgemeines zum Layout

Erster Schritt nach dem Erstellen eines neuen *Window* oder einer neuen *Page* ist die Auswahl einer passenden Panel-Klasse[1], die das spätere Layout der Seite vorgibt. WPF bietet hier eine reiche Auswahl an, die folgende Tabelle zeigt einige wichtige Panels:

Layout-Control	Kurzbeschreibung
Canvas	In diesem Panel können Sie eine absolute Positionierung realisieren.
Dockpanel	Controls können im Panel andocken (*Top*, *Bottom, Left, Right, Fill).*
Grid	Ein Tabellen-Layout wie es auch bei HTML-Seiten verwendet wird.
Stackpanel	Controls werden übereinander (vertikal) oder nebeneinander (horizontal "gestapelt". Die Richtung kann vorgegebenen werden.
TabPanel	Entspricht einem *TabControl*, enthält untergeordnete *TabItem*s, die wiederum andere Panel-Controls aufnehmen können.
UniformGrid	Formatiert enthaltene Controls in ein festes Raster aus Zeilen und Spalten, alle Controls sind gleich groß.
ViewBox	Enthaltene Grafiken können verschiedenartig skaliert werden.
WrapPanel	Dieses Panel verhält sich ähnlich wie ein *StackPanel,* allerdings werden die enthaltenen Controls bei Bedarf in die nächste "Zeile" umgebrochen.

Wie Sie sehen, besitzt jedes diese Elemente spezielle Verhaltensweisen für die Anordnung der darin abgelegten Controls. Doch warum brauchen wir überhaupt ein Layout?

Die Antwort findet sich in der Möglichkeit von WPF-Anwendungen, Dialoge frei zu skalieren, d.h., eine Größenänderung des Formulars soll sich auch **sinnvoll** auf die darin enthaltenen Controls auswirken. Absolute Positions- und Größenangaben sind in diesem Zusammenhang ungeeignet, vielmehr soll das Layout durch geschickte Auswahl und Parametrierung von Panel-Controls erzeugt werden. Dabei ist es auch möglich, die Panel-Controls ineinander zu verschachteln, um die gewünschten Effekte zu erreichen.

[1] Standardmäßig ist in Visual Studio bereits ein *Grid* enthalten (keine Zeile/Spalten).

BEISPIEL 12.13: Darstellung eines einfachen Taschenrechners (auszugsweise)

XAML

Zunächst das komplette Fenster definieren:

```
<Window x:Class="WpfApplication2.Window2"
...
```

Ein *Grid* mit zwei Zeilen, einer Spalte teilt die Grundfläche im Verhältnis 1:4:

```
  <Grid>
   <Grid.RowDefinitions>
     <RowDefinition Height="*" />
     <RowDefinition Height="4*" />
   </Grid.RowDefinitions>
```

In die obere Zelle des Grids wird eine Textbox zur Ausgabe der Werte eingefügt, diese definiert ihre Randabstände über das Attribut *Margin*:

```
   <TextBox Grid.Row="0" Margin="4,4,4,10" Name="textBox1" />
```

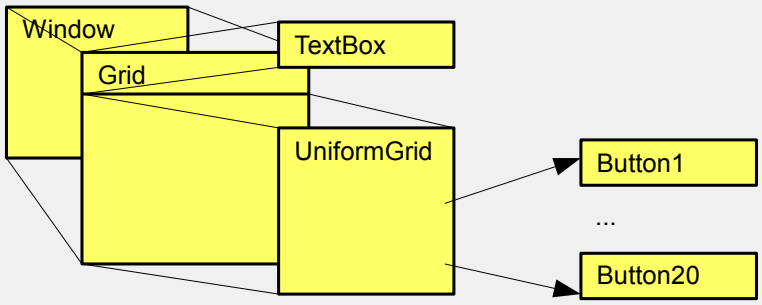
In die untere Zelle des Grids fügen wir ein *UniformGrid* ein, enthaltene Steuerelemente werden in ihrer Größe automatisch an ein per *Columns* bzw. *Rows* festgelegtes Raster angepasst:

```
   <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
```

Was fehlt sind unsere Schaltflächen, die durch das *UniformGrid* automatisch skaliert und positioniert werden. Wir können also auf all diese Angaben verzichten:

```
     <Button  Name="button1" >7</Button>
     <Button  Name="button2" >8</Button>
...
     <Button  Name="button20" >=</Button>
   </UniformGrid>
 </Grid>
</Window>
```

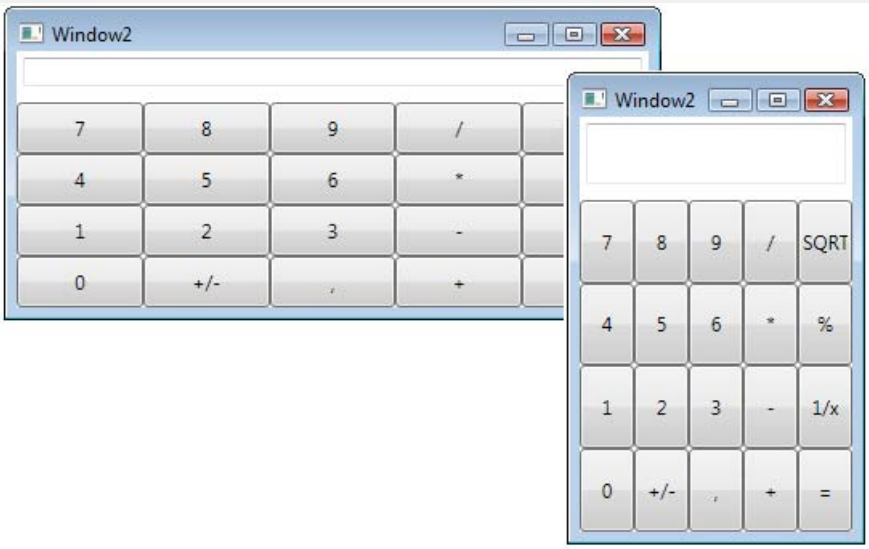
Dieses noch recht einfache Beispiel hat bereits vier Hierarchieebenen, wie sie in der folgenden Abbildung noch einmal gezeigt werden:



BEISPIEL 12.13: Darstellung eines einfachen Taschenrechners (auszugsweise)

Ergebnis

Zur Laufzeit können Sie das Fenster jetzt beliebig vergrößern/verkleinern bzw. im Seitenverhältnis verändern, das Grundlayout des Fensters bleibt erhalten:



HINWEIS: Vergessen Sie in diesem Zusammenhang nicht die Regel, dass Controls in einigen Panels ohne explizite Größenangabe ihre Abmessungen nach dem Inhalt (Content) bestimmen.

12.3.2 Positionieren von Steuerelementen

Wer sich obiges Beispiel bereits genauer angesehen hat, wird festgestellt haben, dass wir trotz allem einige absolute Maßangaben im XAML-Code versteckt haben. Die Rede ist hier von der *TextBox,* deren äußere Ränder wir mit *Margin* fest definiert haben.

Was sind das überhaupt für Maßangaben?

Um was für Maßangaben bzw. Einheiten (Pixel, Inch ...) handelt es sich eigentlich? Die Lösung: In WPF werden Koordinatenangaben in geräteunabhängigen Pixeln angegeben! Wer jetzt auch nicht schlauer ist, dem sei gesagt, dass Microsoft die Größe eines derartigen Pixels mit genau 1/96 Zoll festgelegt hat.

Dieser Wert leitet sich aus der "angenommenen" Standard-Bildschirmauflösung von 96 DPI (Pixel/Inch) ab, im Idealfall entspricht also 1 logischer Pixel einem Pixel auf dem Bildschirm. Dass diese Annahme für absolute Maßangaben ziemlicher Blödsinn ist, dürfte spätestens nach einem Wechsel der Bildschirmauflösung klar sein, ein Pixel ist jetzt viel größer/kleiner als vorher.

Denken sie also nicht allzu viel darüber nach und nutzen Sie diesen Wert einfach so wie die allseits bekannten Pixel.

Wer Probleme mit der Umrechnung zwischen dieser schönen neuen Einheit und den konventionellen Einheiten hat, kann sich mit folgenden Umrechungsfaktoren behelfen:

Masseinheit	Von WPF-Pixel nach ...	Von ... in WPF-Pixel
Inch	0,01041666	96
Millimeter	0,26458333	3,779527
Point	0,75	1,333333

Top/Left/Width/Height

Wenn Sie sich ein Control bzw. dessen Eigenschaften näher ansehen, werden Sie schnell feststellen, dass weder *Top* noch *Left* vorhanden sind. Wozu auch, in den meisten Layout-Controls findet sowieso eine automatische Positionierung statt.

Einzige Ausnahme: das *Canvas*-Control, wo eine absolute Positionierung möglich ist. In diesem Fall helfen Ihnen so genannte "angehängte Eigenschaften" (*Attached Properties*) weiter. Diese beziehen sich jeweils auf das übergeordnete Control, in diesem speziellen Fall auf das *Canvas*-Control.

BEISPIEL 12.14: Positionieren einer Schaltfläche in einem *Canvas*-Control

C#

```
<Canvas Name="canvas1" >
  <Button Canvas.Left="74" Canvas.Top="70" Height="45" Name="button1"
          Width="89">Beschriftung
  </Button>
</Canvas>
```

HINWEIS: Mehr zum *Canvas* finden Sie ab Seite 606.

Zu *Width* und *Height* brauchen wir Ihnen sicher nicht viel zu sagen, beachten Sie jedoch, dass in den meisten Fällen die Breite/Höhe des Controls durch den umgebenden Container bestimmt wird. Dies ist auch im Sinne einer layout-orientierten Programmierung, in der die Gestaltung vom übergeordneten Layout und nicht vom einzelnen Control bestimmt wird.

MinWidth/MaxWidth/MinHeight/MaxHeight

Mit diesen Attributen/Eigenschaften können Sie Mindestgrößen für das Control vorgeben. Sind diese nicht realisierbar (zu wenig Platz), wird das Control abgeschnitten, ist mehr Platz, wird das Control in seiner Größe beschränkt.

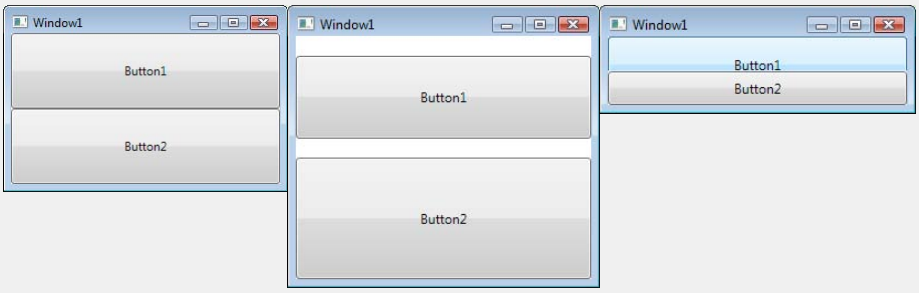
BEISPIEL 12.15: Höhenbeschränkung einer Schaltfläche

XAML

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Button Grid.Row="0" MinHeight="50" MaxHeight="75">Button1</Button>
  <Button Grid.Row="1" >Button2</Button>
</Grid>
```

Ergebnis

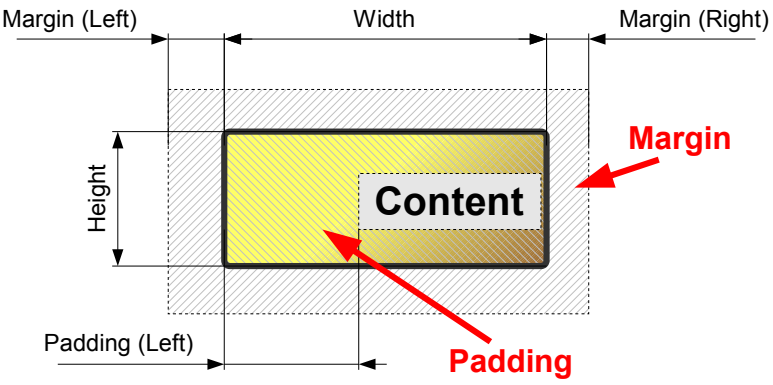
Defaultanzeige | *MaxHeight* dominiert (es entstehen Freiflächen) | *MinHeigh* dominiert (der Button wird beschnitten)



Margin/Padding

Im Zusammenhang mit der Ausrichtung von Controls sind auch zwei Eigenschaften interessant, die zum einen den Abstand des Controls zum umgebenden Elternfenster/-Control angeben (Margin), zum anderen den Abstand des Control-Contents zu den Control-Außengrenzen (Padding).

Die folgende Abbildung zeigt die entsprechenden Bereiche:



Beide Eigenschaften können Sie vollständig (4 Werte) oder auch vereinfacht angeben (1 Wert, 2 Werte).

Werte	Beispiel	Bedeutung
1 Wert	*Margin="5"*	der Wert gilt für alle Ränder
2 Werte	*Margin="5,10"*	der erste Wert bestimmt den linken und rechten Rand, der zweite Wert bestimmt den oberen und unteren Rand
4 Werte	*Margin="1,5,10,20"*	Einzelangaben für linken, oberen, rechten, unteren Rand

HINWEIS: Die Angabe von 3 oder mehr als vier Werten führt zu einem Laufzeitfehler.

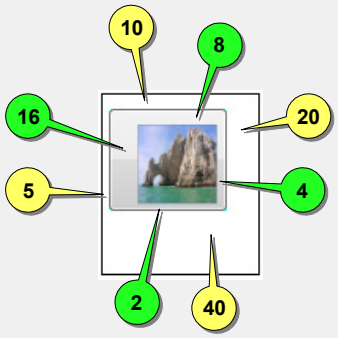
BEISPIEL 12.16: Verwendung von *Padding* und *Margin* für die Gestaltung einer Schaltfläche

XAML

```
...
  <Button  Margin="5,10,20,40" Padding="16,8,4,2" Name="button2">
    <Image Width="50" Height="50" Source="Bilder/Cabo.jpg" />
  </Button>
...
```

Ergebnis

Die Ausgabe mit Angabe der Abstände:



HorizontalAlignment/VerticalAligment

Vielleicht wundern Sie sich an dieser Stelle, warum wir uns mit der horizontalen bzw. vertikalen Ausrichtung von Controls beschäftigen, wenn diese doch meist in die übergeordneten Layout-Controls hineinskaliert werden.

Der Grund für dieses Verhalten ist die Defaulteinstellung für beide Eigenschaften, diese ist mit *Stretch* festgelegt. Alternativ können Sie für *HorizontalAlignment* auch *Left, Right, Center* auswählen bzw. für *VerticalAlignment* die Werte *Top*, *Bottom* oder *Center.*

Ist ein Wert ungleich *Stretch* festgelegt, bestimmt der enthaltene Content des Controls die Größe bzw. die Angaben von *Width* und *Height*.

12.3.3 Canvas

Mit diesem Control haben Sie die Möglichkeit, die klassische Variante für die Positionierung von Steuerelementen zu realisieren. Dazu stehen Ihnen mit *Top, Left, Right, Bottom* vier angehängte Eigenschaften zur Verfügung.

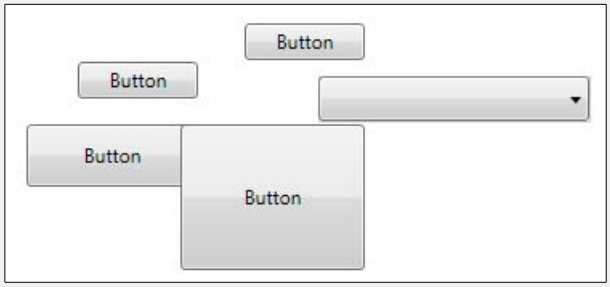
BEISPIEL 12.17: Freies Positionieren von Controls in einem *Canvas* mit *Left* und *Top*

XAML

```
<Canvas Height="196" Width="396">
  <Button Canvas.Left="56" Canvas.Top="46" Height="23" Width="75">Button</Button>
  <Button Canvas.Left="24" Canvas.Top="85" Height="39" Width="107">Button</Button>
  <Button Canvas.Left="160" Canvas.Top="22" Height="23" Width="75">Button</Button>
  <Button Canvas.Left="120" Canvas.Top="85" Height="91" Width="115">Button</Button>
  <ComboBox Canvas.Left="206" Canvas.Top="55" Height="28" Width="169" />
</Canvas>
```

Ergebnis

Die Laufzeitansicht:



Beachten Sie, dass bei einer Verankerung am rechten oder unteren Rand, im Gegensatz zur *Anchor*-Eigenschaft bei den Windows Forms, die Breite und Höhe des Controls unverändert bleibt:

BEISPIEL 12.18: Auswirkung einer Fenstergrößenänderung bei Verankerung an der rechten unteren Ecke:

XAML

```
<Canvas >
  <Button Canvas.Right="10" Canvas.Bottom= "20" Height="23" Width="75">Button</Button>
</Canvas>
```

Ergebnis

12.3.4 StackPanel

Das *StackPanel* ermöglicht das einfache "Stapeln" der enthaltenen Controls. Die Richtung dieses Stapels können Sie über die *Orientation*-Eigenschaft steuern.

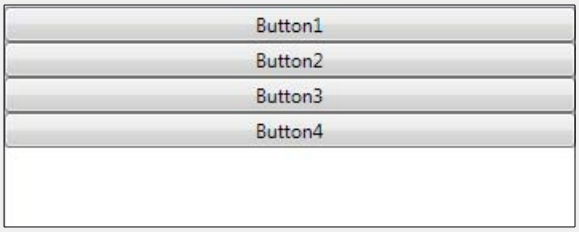
BEISPIEL 12.19: *StackPanel* mit vertikaler Ausrichtung (Default)

XAML

```
<StackPanel>
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
</StackPanel>
```

Ergebnis

Da keine der enthaltenen Schaltflächen eine Größenangabe enthält, werden die Schaltflächen automatisch an die Breite des *StackPanel*s angepasst, die Höhe ergibt sich aus dem Content der jeweiligen Schaltfläche, wie es auch die folgende Abbildung zeigt:



Änderungen der Breite des *StackPanel*s wirken sich unmittelbar auf die Breite der enthaltenen Controls aus. Wird die Defaultgröße der enthaltenen Controls unterschritten, werden diese abgeschnitten.

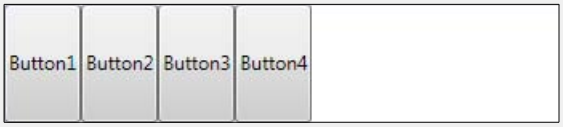
Bei horizontaler Ausrichtung kehrt sich das Verhalten um, jetzt wird die Höhe der Controls durch das *StackPanel* bestimmt, die Breite bestimmt sich aus dem Content.

BEISPIEL 12.20: Horizontale Ausrichtung im *StackPanel*

XAML

```
<StackPanel Orientation="Horizontal" >
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
</StackPanel>
```

Ergebnis



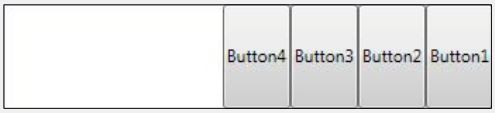
HINWEIS: Über die Eigenschaft *FlowDirection* steuern Sie bei horizontaler Ausrichtung die Fließrichtung (*RightToLeft, LeftToRight*), d.h. die Anzeigereihenfolge.

BEISPIEL 12.21: Änderung der *FlowDirection*

XAML

```
<StackPanel FlowDirection="RightToLeft" Orientation="Horizontal">
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
</StackPanel>
```

Ergebnis



12.3.5 DockPanel

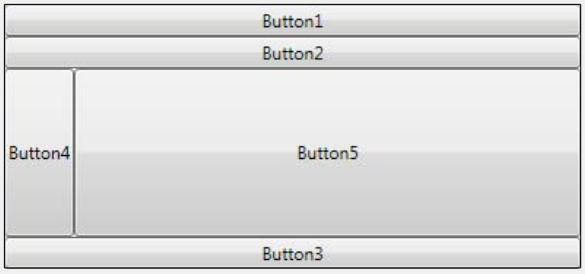
Eines der wichtigsten Controls für das Erzeugen Explorer-ähnlicher Oberflächen dürfte das *DockPanel* sein. Hier können Sie festlegen, an welcher Seite des *DockPanel*s die enthaltenen Controls ausgerichtet werden sollen. Die Ausrichtung wird bei jedem eingelagerten Control über die angehängte Eigenschaft *DockPanel.Dock* bestimmt.

BEISPIEL 12.22: Einige Schaltflächen in einem *DockPanel* ausrichten

XAML

```
<DockPanel >
  <Button DockPanel.Dock="Top">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis



Drei Regeln lassen sich bereits aus obigem Beispiel erkennen:

- Controls mit gleicher Ausrichtung werden gestapelt (*Button1, Button2*).
- Die Reihenfolge der Controls bestimmt, welches Control "dominant" ist (siehe folgendes Beispiel).
- Das letzte Control in der Liste füllt den verbliebenen Platz komplett aus, egal welche Angaben gemacht wurden (*Button5).*

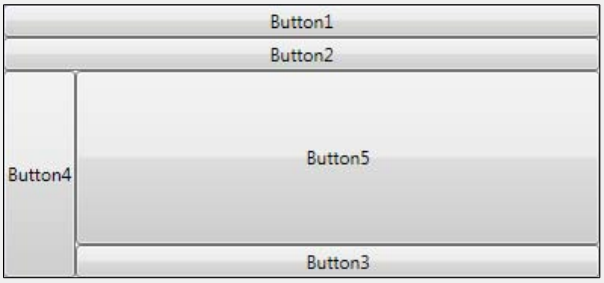
BEISPIEL 12.23: Einfluss der Reihenfolge auf die Anzeige (wir tauschen lediglich *Button3* und *Button4* miteinander)

XAML

```
<DockPanel >
  <Button DockPanel.Dock="Top">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  <Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis

Wie Sie sehen, ist jetzt *Button4* dominanter und bestimmt damit auch die Breite von *Button3* und *Button5:*



Soll das letzte enthaltene Control nicht per Default den gesamten verbliebenen Platz ausfüllen, können Sie dieses Verhalten mit *LastChildFill=False* abschalten.

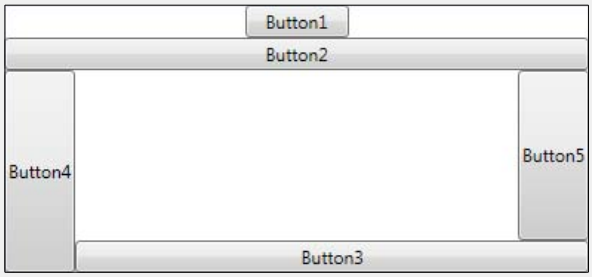
BEISPIEL 12.24: Änderung des Füllverhaltens und der Breite von Elementen

XAML

```
<DockPanel LastChildFill="False">
  <Button DockPanel.Dock="Top" Width="70">Button1</Button>
  <Button DockPanel.Dock="Top">Button2</Button>
  <Button DockPanel.Dock="Left">Button4</Button>
  <Button DockPanel.Dock="Bottom">Button3</Button>
  <Button DockPanel.Dock="Right">Button5</Button>
</DockPanel>
```

Ergebnis

Geben Sie bei den enthaltenen Controls *Height* oder *Width* an, überschreiben diese Werte die automatisch vorgegebenen Werte des *DockPanel*s (*Button1*):



Im obigen Beispiel wurde *Button1* per Default horizontal zentriert dargestellt. Mit *HorizontalAligment* bzw. *VerticalAligment* können Sie dieses Verhalten ändern.

HINWEIS: Controls in einem *DockPanel* können sich nicht überlappen, notfalls werden diese abgeschnitten.

12.3.6 WrapPanel

Hier haben wir es mit einem nahen Verwandten des *StackPanels* zu tun, das Verhalten ist recht ähnlich, mit dem Unterschied, dass bei fehlendem Platz die enthaltenen Elemente in die nächste "Zeile" umgebrochen werden.

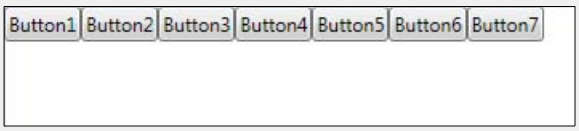
BEISPIEL 12.25: Ein *WrapPanel* mit Buttons füllen

XAML

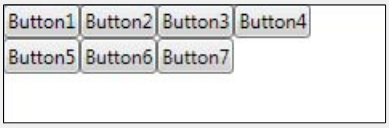
```
<WrapPanel>
  <Button>Button1</Button>
  <Button>Button2</Button>
  <Button>Button3</Button>
  <Button>Button4</Button>
  <Button>Button5</Button>
  <Button>Button6</Button>
  <Button>Button7</Button>
</WrapPanel>
```

Ergebnis

Die erste Ansicht:



Eine Breitenänderung des *WrapPanel*s führt zu folgender Ansicht:



Ist nicht mehr genügend Platz für einen Umbruch vorhanden, werden die betreffenden Elemente abgeschnitten.

12.3.7 UniformGrid

Bevor wir uns dem wesentlich komplexeren Verwandten zuwenden, wollen wir noch einen kurzen Blick auf das *UniformGrid* werfen. Dieses dient dem einfachen Ausrichten von Elementen, wenn diese alle die gleiche Größe erhalten und in einer Rasterstruktur angeordnet werden sollen.

Die Verwendung ist relativ einfach: Definieren Sie über die Eigenschaften *Columns* und *Rows* zunächst die Anzahl der Zeilen und Spalten und fügen Sie die gewünschten Elementen ein.

HINWEIS: Eine gezielte Zuordnung zu den einzelnen Gridzellen ist nicht möglich, hier entscheidet die Reihenfolge der Definition über die Position im *UniformGrid*. Die Zellen werden von links nach rechts und dann von oben nach unten gefüllt.

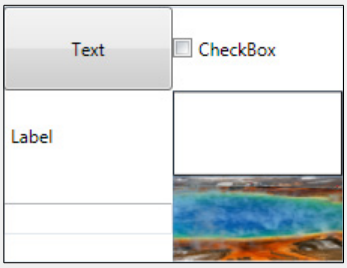
BEISPIEL 12.26: Verwendung des *UniformGrid*

XAML

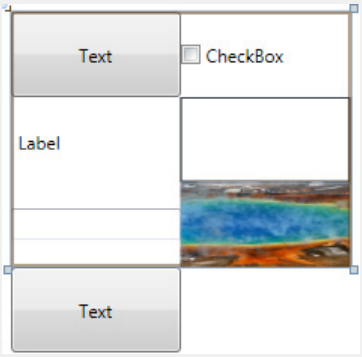
```
<UniformGrid Name="uniformGrid1" Columns="2" Rows="3" Grid.Row="1" Height="170"
             Width="226">
  <Button>Text</Button>
  <CheckBox Height="16" Name="checkBox1" Width="120">CheckBox</CheckBox>
  <Label Height="23" Name="label1" Width="120">Label</Label>
  <Rectangle Name="rectangle1" Stroke="Black" />
  <TextBox Height="21" Name="textBox1" />
  <Image Name="image1" Source="Crater.jpg" />
</UniformGrid>
```

Ergebnis

Das angezeigte *UniformGrid:*



HINWEIS: Ist für das *UniformGrid* eine Höhe bzw. Breite vorgegeben, werden Kind-Elemente auch dann dargestellt, wenn sie die Anzahl der vorgegebenen Zeilen und Spalten überschreiten.



12.3.8 Grid

Hier haben wir es mit einem recht komplexen Control zu tun, bei dem Kind-Elemente in Zeilen und Spalten angeordnet werden können. Im Unterschied zum *UniformGrid* ist das *Grid* jedoch wesentlich flexibler, was die Definition von Zeilen und Spalten anbelangt. Auch die enthaltenen Elemente können freier positioniert werden, es ist sogar möglich, Elemente zellübergreifend zu definieren. Doch der Reihe nach:

Definition des Grundlayouts

Fügen Sie ein neues *Grid* in ein *Window* ein, verfügt dieses zunächst nur über eine einzige Zelle. Neue Zeilen bzw. Spalten definieren Sie über einen gesonderten Bereich *RowDefinitions* bzw. *ColumnDefinitions*.

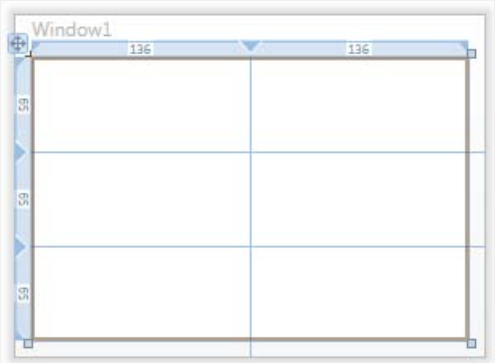
BEISPIEL 12.27: Ein *Grid* mit zwei Spalten und drei Zeilen definieren

XAML

```
<Window x:Class="WpfApplication2.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="215" Width="294"  >
 <Grid>
   <Grid.RowDefinitions>
     <RowDefinition/>
     <RowDefinition/>
     <RowDefinition/>
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
     <ColumnDefinition/>
     <ColumnDefinition/>
   </Grid.ColumnDefinitions>
 </Grid>
</Window>
```

Ergebnis

Wie Sie sehen, sind bei dieser Art der Definition alle Zellen gleich groß, vergrößern Sie das *Grid*, werden die Spalten und Zeilen im gleichen Verhältnis vergrößert.



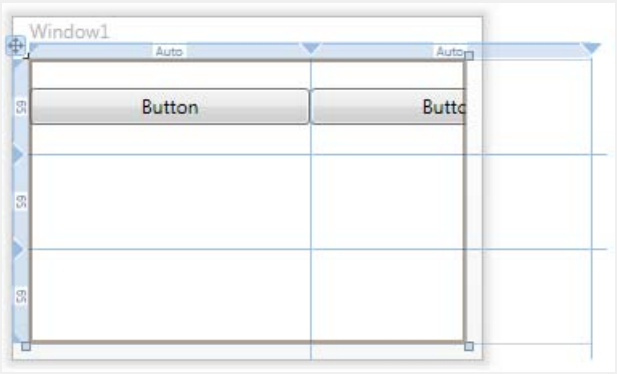
Für die Definition spezifischer Zeilenhöhen bzw. Spaltenbreiten hat sich Microsoft einige Varianten ausgedacht:

BEISPIEL 12.28: Spaltenbreite entsprechend dem Inhalt festlegen (das breiteste bzw. höchste Element bestimmt die Breite bzw. Höhe)

XAML

```
<Grid>
...
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
  <Button Grid.Column="0" Height="23" Name="button1" Width="175"
          Grid.Row="0">Button</Button>
  <Button Grid.Column="1" Height="23" Name="button2" Width="175"
          Grid.Row="0">Button</Button>
</Grid>
```

Ergebnis



BEISPIEL 12.29: Festlegen einer bestimmten Breite

XAML

```
<Grid>
...
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100"/>
    <ColumnDefinition Width="Auto"/>
  </Grid.ColumnDefinitions>
...
</Grid>
```

BEISPIEL 12.30: Festlegen eines Anteils am verfügbaren Platz

XAML

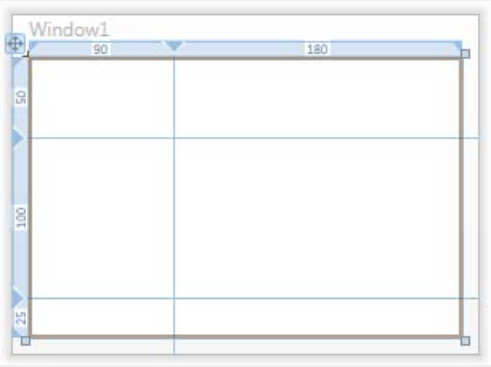
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="2*"/>
    <RowDefinition Height="0.5*"/>
```

BEISPIEL 12.30: Festlegen eines Anteils am verfügbaren Platz

```
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="2*"/>
  </Grid.ColumnDefinitions>
</Grid>
```

Ergebnis

In diesem Fall kommen Sie um etwas Rechnerei nicht herum. Für obiges Beispiel gilt: Es gibt 3,5 Zeilenanteile (entspricht 175 Einheiten) und 3 Spaltenanteile (entspricht 270 Einheiten). Die daraus resultierenden Spaltenbreiten bzw. Zeilenhöhen können Sie der folgenden Laufzeitansicht entnehmen:



Wie Sie sehen, können die Zellgrößen im *Grid* recht flexibel definiert werden.

Zuordnen von Kind-Elementen

Die Zuordnung der Kindelemente zu den einzelnen Gridzellen erfolgt über die angehängten Eigenschaften *Grid.Column* und *Grid.Row*, die Sie bei jedem der Kindelemente setzen können.

BEISPIEL 12.31: Erstellen einer einfachen Oberfläche

XAML

Zunächst definieren wir das *Grid* und dessen Hintergrundfarbe:

```
<Grid Background="AliceBlue">
```

Die Zeilen festlegen:

```
  <Grid.RowDefinitions>
    <RowDefinition Height="*"/>
    <RowDefinition Height="2*"/>
    <RowDefinition Height="0.5*"/>
  </Grid.RowDefinitions>
```

Die Spalten festlegen:

```
  <Grid.ColumnDefinitions>
```

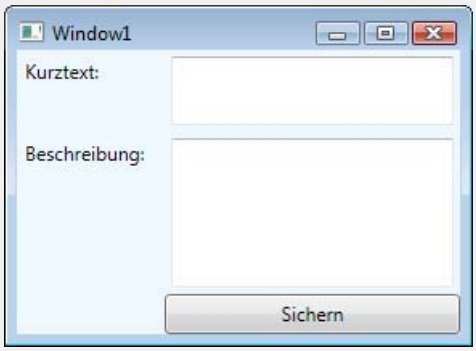
BEISPIEL 12.31: Erstellen einer einfachen Oberfläche

```
    <ColumnDefinition Width="*"/>
    <ColumnDefinition Width="2*"/>
  </Grid.ColumnDefinitions>
```

Und hier kommen die Inhalte:

```
  <Label Grid.Column="0" Grid.Row="0">Kurztext:</Label>
  <Label Grid.Column="0" Grid.Row="1">Beschreibung:</Label>
  <TextBox Grid.Column="1" Grid.Row="0" Margin="4" Name="textBox1" />
  <TextBox Grid.Column="1" Grid.Row="1" Margin="4" Name="textBox2" AcceptsReturn="True"/>
  <Button Grid.Column="1" Grid.Row="2"  Name="button1">Sichern</Button>
</Grid>
```

Ergebnis



HINWEIS: Für das Ausrichten der Inhalte können Sie die Eigenschaften *VerticalAlignment* und *HorizontalAlignment* verwenden.

Doch was ist, wenn sich ein Element über zwei oder drei Spalten bzw. Zeilen erstrecken soll? Auch das ist kein Problem, verwenden Sie dafür die Eigenschaften *Grid.ColumnSpan* bzw. *Grid.RowSpan*, um für das jeweilige Element mehrere Spalten/Zeilen zusammenzufassen.

BEISPIEL 12.32: Die Schaltfläche aus obigem Beispiel soll sich über zwei Spalten erstrecken

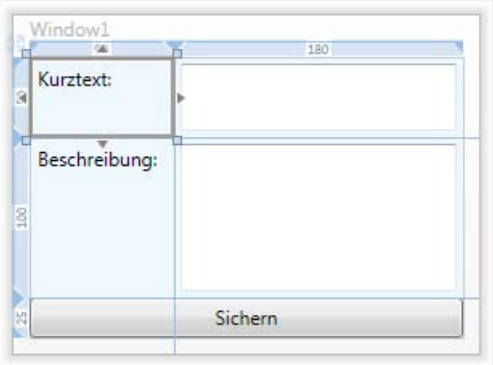
XAML

```
<Grid Background="AliceBlue">
...
  <Button Grid.Column="0" Grid.Row="2"  Grid.ColumnSpan="2" Name="button1">Sichern</Button>
</Grid>
```

Ergebnis

In der Entwurfsansicht können Sie deutlich sehen, dass sich die Schaltfläche jetzt über die Spaltentrennlinie erstreckt (siehe folgende Abbildung):

BEISPIEL 12.32: Die Schaltfläche aus obigem Beispiel soll sich über zwei Spalten erstrecken



Verwendung des GridSplitters

Möchten Sie zur Laufzeit Einfluss auf die Spaltenbreiten bzw. Zeilenhöhen nehmen, können Sie einen so genannten *GridSplitter* in eine oder auch mehrere Zellen (*ColumnSpan, RowSpan*) einfügen. Dessen Breite bestimmen Sie mit *Width,* die Farbe können Sie über das *Background*-Attribut festlegen.

HINWEIS: Per Default ist der *GridSplitter* am rechten Rand der Zelle verankert, Sie können dieses Verhalten aber auch über die *HorizontalAlignment*-Eigenschaft ändern.

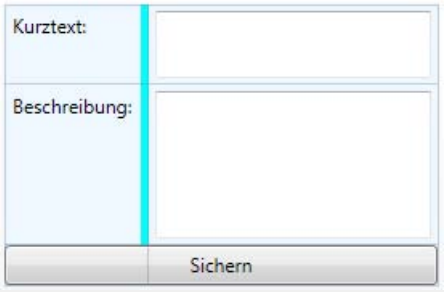
BEISPIEL 12.33: Verwendung *GridSplitter* für obiges Beispiel

XAML

```
<Grid Background="AliceBlue">
...
  <GridSplitter Grid.Column="0" Grid.RowSpan="2" HorizontalAlignment="Right" Width="5"
                Background="Cyan" />
</Grid>
```

Ergebnis

Die Entwurfsansicht zeigt bereits, dass der *GridSplitter* nur in den ersten beiden Zeilen des Grids angezeigt wird:



HINWEIS: Der *GridSplitter* überdeckt den Inhalt der jeweiligen Zelle etwas, legen Sie also für die enthaltenen Elemente einen entsprechend großen Randabstand fest (*Margin*).

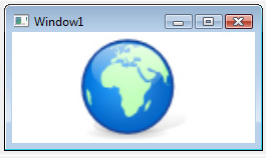
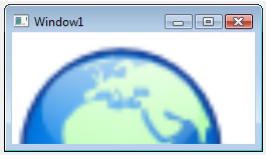
12.3.9 ViewBox

Geht es um die Anzeige von Grafiken in bestimmten Seitenverhältnissen bzw. mit automatischer Anpassung an das übergeordnete Element, ist die *ViewBox* die erste Wahl.

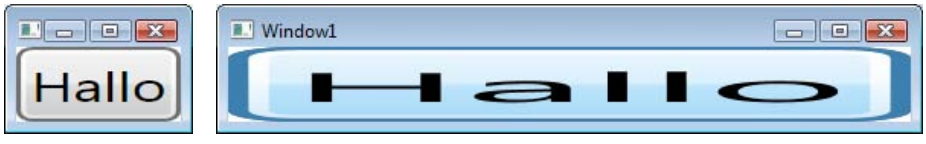
BEISPIEL 12.34: Platzieren Sie einfach die gewünschte Grafik im Clientbereich und legen Sie per *Stretch*-Attribut das Verhalten fest.

XAML
```
<Viewbox Name="viewbox1" Stretch="UniformToFill" >
  <Image Source="browser.png"/>
</Viewbox>
```

Die Tabelle zeigt die Auswirkung der *Stretch*-Eigenschaft auf das Aussehen der enthaltenen Grafik:

Stretch	Beispiel	Beschreibung
None		Grafik wird in Originalgröße angezeigt. Reicht der Platz nicht, wird die Grafik abgeschnitten. Ist die Grafik kleiner als die *ViewBox*, wird die Grafik per Default zentriert.
Fill		Die Grafik wird ohne Rücksicht auf die Proportionen in die *ViewBox* skaliert.
Uniform		Die Grafik wird proportional skaliert, sodass diese vollständig in der Clientfläche angezeigt wird.
UniformToFill		Die Grafik wird proportional skaliert, jedoch so, dass die *ViewBox* vollständig gefüllt ist. Hierbei werden meist Teile der Grafik abgeschnitten

Dass auch beliebige andere Elemente problemlos mit der *ViewBox* skaliert werden können, zeigt das folgende Beispiel eines Buttons:



HINWEIS: Da es sich um Vektorgrafik handelt, sind auch starke Vergrößerungen ohne Qualitätsverlust möglich.

12.3.10 TextBlock

Im Gegensatz zu den bisher vorgestellten Elementen nimmt der *TextBlock* eine Sonderstellung ein. Auch wenn dieser prinzipiell weitere Steuerelemente aufnehmen kann, ist doch seine Hauptaufgabe die Ausgabe von formatierten Texten. Damit empfiehlt sich dieses Control als "großer Bruder" des guten alten *Label*-Controls, mit wesentlich erweiterten Möglichkeiten.

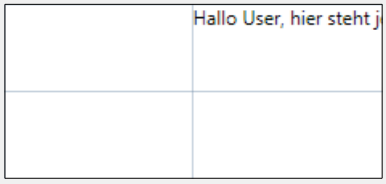
BEISPIEL 12.35: Ausgabe von Text in einer Gridzelle mit Hilfe eines *TextBlock*-Elements

XAML

```
...
 <Grid>
   <Grid.ColumnDefinitions>
     <ColumnDefinition/><ColumnDefinition/>
   </Grid.ColumnDefinitions>
   <Grid.RowDefinitions>
     <RowDefinition/><RowDefinition/>
   </Grid.RowDefinitions>
   <TextBlock Grid.Column="1" Grid.Row="0">
       Hallo User, hier steht jede Menge Text!
     </TextBlock>
 </Grid>
...
```

Ergebnis

Ein Ergebnis ist bereits zu sehen, dürfte aber noch nicht ganz den Erwartungen entsprechen, da der Text nicht umgebrochen wird



BEISPIEL 12.36: Umbrechen des Textes mit dem Attribut *TextWrapping*

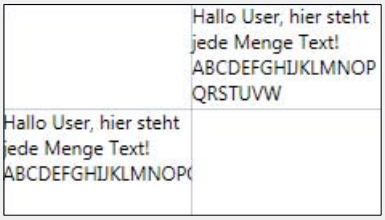
XAML

```
...
   <TextBlock TextWrapping="Wrap" Grid.Column="1" Grid.Row="0">
       Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
     </TextBlock>
   <TextBlock TextWrapping="WrapWithOverflow" Grid.Column="0" Grid.Row="1">
     Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
   </TextBlock>
...
```

BEISPIEL 12.36: Umbrechen des Textes mit dem Attribut *TextWrapping*

Ergebnis

Die beiden möglichen Umbruch-Varianten unterscheiden sich nur beim Umbruch langer Wörter, die nicht in eine Zeile passen (*Wrap* trennt das Wort, *WrapWithOverflow* schneidet es ab):



Textformatierungen

Der auffälligste Unterschied zum *Label* besteht in den umfangreichen Formatierungsmöglichkeiten, die ein *TextBlock* anbietet:

BEISPIEL 12.37: Einfache Vorgabe eines Textformats für den gesamten *TextBlock*

XAML

```
...
  <TextBlock FontFamily="Arial" FontSize="18" FontWeight="Bold" TextWrapping="Wrap">
      Hallo User, hier steht jede Menge Text! ABCDEFGHIJKLMNOPQRSTUVW
  </TextBlock>
...
```

Ergebnis

BEISPIEL 12.38: Spezielle Textformatierungen im *TextBlock*

XAML

```
...
  <TextBlock FontSize="24" TextWrapping="Wrap">
      Hallo<Bold>User</Bold>, hier steht <Underline>jede Menge</Underline>
      <Italic>Text</Italic>! ABCDEFGHIJKLMNOPQRSTUVW
  </TextBlock>
...
```

BEISPIEL 12.38: Spezielle Textformatierungen im *TextBlock*

Ergebnis

Der formatierte Text:

Besonderheit von Leerzeichen/Zeilenumbrüchen

Auf ein Thema der Textausgabe müssen wir unbedingt noch eingehen, da es sich hier um eine Besonderheit des XAML-Compilers handelt. Die Rede ist von der Ausgabe mehrfacher Leerzeichen und von Zeilenumbrüchen, die standardmäßig nicht berücksichtigt werden.

Mit dem Attribut *xml:space* können Sie das Verhalten des Compilers beeinflussen, sodass auch mehrfache Leerzeichen und Zeilenumbrüche richtig interpretiert werden.

HINWEIS: Doch Achtung: In diesem Fall werden auch die Formatierungen im XAML-Quelltext (Einrückungen) berücksichtigt.

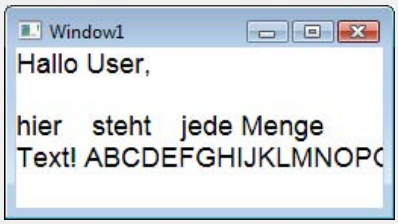
BEISPIEL 12.39: Ausgabe mehrfacher Leerzeichen und Zeilenumbrüche

XAML

```
<TextBlock xml:space="preserve" FontFamily="Arial" FontSize="18">Hallo User,

hier    steht    jede Menge
Text! ABCDEFGHIJKLMNOPQRSTUVW
</TextBlock>
```

Ergebnis

Die Ausgabe (beachten Sie, dass Wrapping nicht aktiviert ist):



HINWEIS: Zeilenumbrüche könne Sie auch mit einem <*LineBreak/*>-Element erzeugen.

Textausrichtung

Neben dem Textformat ist meist auch eine spezielle Ausrichtung der Texte gewünscht. Der *TextBlock* verwendet dafür, wie wohl nicht anders zu erwarten, das Attribut *TextAligment*. Die möglichen Werte sind *Left, Right, Center* (mittig zentriert), *Justify* (Blocksatz).

12.4 Das WPF-Programm

Da zeigen wir Ihnen im vorhergehenden Abschnitt schon diverse WPF-Elemente und kümmern uns gar nicht um das eigentliche Programmgerüst! Das wollen wir nun nachholen, bevor wir uns mit weiteren Details der WPF-Controls beschäftigen.

Wie bereits im Abschnitt 12.2.5 erwähnt, müssen Sie zwischen zwei grundsätzlichen WPF-Anwendungstypen unterscheiden:

- WPF-Anwendung
- WPF-Browseranwendung

Erstere ist eine Ansammlung von einzelnen Windows, wie Sie es auch aus den altgedienten Windows Forms-Anwendungen kennen[1], WPF-Browseranwendungen verwenden statt einzelner Fenster so genannte Pages, zwischen denen navigiert werden kann (ähnlich ASP.NET).

HINWEIS: Im Folgenden wollen wir uns aus Platzgründen auf den WPF-Anwendungstyp beschränken.

12.4.1 Die App-Klasse

Auf die in der Datei *App.xaml.cs* abgeleitete *App*-Klasse sind wir bereits auf Seite 589 eingegangen. Eine zugehörige Instanz dieser Klasse können Sie zur Laufzeit über die statische Methode *App.Current* abrufen.

Interessant ist diese Klasse vor allem für den Lebenszyklus des Programms:

- Festlegen des Startobjekts
- Auswerten von Kommandozeilenparametern
- Beenden der Anwendung
- Auswerten von Anwendungsereignissen

Wir wollen uns die einzelnen Punkte im Folgenden etwas näher ansehen.

[1] SDI-Anwendung, MDI-Anwendungen werden noch nicht unterstützt.

12.4.2 Das Startobjekt festlegen

Wie schon erwähnt, wird das Hauptprogramm der WPF-Anwendung automatisch vom Visual Studio-Compiler erzeugt. Doch wo legen wir dann das Startobjekt bzw. das Startfenster fest?

Hier hilft uns die Datei *App.xaml* weiter, in dieser wird mit dem *StartupUri*-Attribut das erste Fenster festgelegt:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
    <Application.Resources>
    </Application.Resources>
</Application>
```

Diesen Eintrag können Sie ohne Probleme auf ein beliebiges anderes Fenster festlegen. Der Compiler nutzt diese Information für die später automatisch erzeugte Methode *InitializeComponent:*

```
[DebuggerNonUserCode]
public void InitializeComponent()
{
    base.StartupUri = new Uri("Window1.xaml", UriKind.Relative);
}
```

HINWEIS: Sie können diesen Wert auch erst zur Laufzeit ändern, überschreiben Sie dazu beispielsweise die *OnStartup*-Methode.

BEISPIEL 12.40: Zur Laufzeit ein Startformular festlegen (*App.xaml.cs*)

C#

```
...
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            if (e.Args.Count() == 0)
                this.StartupUri = new Uri("Window2.xaml", UriKind.Relative);
            else
                this.StartupUri = new Uri("Window1.xaml", UriKind.Relative);
        }
    }
```

BEISPIEL 12.41: Alternative zum Überschreiben der *OnStartup*-Methode

XAML

Alternativ können Sie auch das entsprechende Ereignis *Startup* verwenden. Fügen Sie dazu in der Datei *App.xaml* einen entsprechenden Tag hinzu:

```
<Application x:Class="WpfApplication2.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

BEISPIEL 12.41: Alternative zum Überschreiben der *OnStartup*-Methode

```
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            StartupUri="MainWindow.xaml" Startup="Application_Startup">
...
    <Application.Resources>
    </Application.Resources>
</Application>
```

C#

Anschließend editieren Sie die Datei *App.xaml.cs*:

```
...
namespace WpfApplication2
{
    public partial class App : Application
    {
        private void Application_Startup(object sender, StartupEventArgs e)
        {
 ...
        }
    }
}
```

HINWEIS: Sie können in diesem Ereignis bzw. in der überschriebenen Methode auch weitere Fenster erzeugen und anzeigen, diese erscheinen zeitgleich mit dem Hauptformular, das per *App.xaml* definiert wurde.

12.4.3 Kommandozeilenparameter verarbeiten

Auch wenn die Verwendung von Kommandozeilenparametern immer mehr aus der Mode kommt, ist es in dem einen oder anderen Fall doch erforderlich.

In der *OnStartUp*-Methode bzw. im *Startup*-Ereignis wird ein Parameter vom Typ *StartupEvent-Args* übergeben. Über dessen Eigenschaft *Args* können Sie auf die Liste der übergebenen Parameter zugreifen.

BEISPIEL 12.42: Überschreiben der Methode *OnStartUp*, um die Kommandozeilenparameter auszuwerten (*App.xaml.cs*) und anzuzeigen:

C#

```
...
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            if (e.Args.Count() == 0)
            {
                MessageBox.Show("Das Programm muss mit Parametern gestartet werden!");
                this.Shutdown();
```

BEISPIEL 12.42: Überschreiben der Methode *OnStartUp*, um die Kommandozeilenparameter auszuwerten (*App.xaml.cs*) und anzuzeigen:

```
            }
            foreach (string arg in e.Args)
                MessageBox.Show(arg);
        }
    }
```

12.4.4 Die Anwendung beenden

Eigentlich eine dumme Aufgabenstellung, werden die meisten denken, wird doch die Anwendung standardmäßig geschlossen, wenn das letzte Fenster vom Anwender geschlossen wird. Dies ist zunächst korrekt, doch dieses Verhalten ist nicht immer gewünscht, und es soll auch Fälle geben, wo die Anwendung aus sich heraus geschlossen wird.

Die Art, wie eine Anwendung beendet wird, beeinflussen Sie über die Eigenschaft *ShutdownMode* des *App*-Objekts. Neben dem Standardwert *OnLastWindowClose* (wenn kein offenes Fenster mehr vorhanden ist) können Sie auch *OnMainWindowClose (*beim Schließen des Startfensters) oder *OnExplicitShutdown* wählen.

Die letzte Variante erwartet den expliziten Aufruf der *Shutdown*-Methode des *Application*-Objekts.

BEISPIEL 12.43: Explizites Beenden der Anwendung per Schaltfläche

C#
```
private void button1_Click(object sender, RoutedEventArgs e)
{
   App.Current.Shutdown();
}
```

12.4.5 Auswerten von Anwendungsereignissen

Für den Programmierer hält die *App*-Klasse noch einige interessante Ereignisse bereit, mit denen er auf bestimmt Anwendungszustände reagieren kann.

Ereignis	Beschreibung
Activated	Die Applikation wird zur Vordergrundanwendung.
Deactivated	Die Applikation ist nicht mehr die Vordergrundanwendung.
DispatcherUnhandledException	Ein nicht behandelter Fehler ist aufgetreten.
Exit	Das Anwendungsende ist ausgelöst.
SessionEnding	Windows wird beendet (Logout oder Shutdown). Sie können den Grund über den Parameter *e.ReasonSessionEnding* ermitteln. Mit *e.Cancel* können Sie versuchen, den Shutdown aufzuhalten, dies ist jedoch nicht in allen Fällen möglich.
Startup	Die Anwendung wurde gestartet.

BEISPIEL 12.44: Eine zentrale Fehlerbehandlung implementieren

C#

In der Datei *App.xaml.cs* erzeugen Sie einen neuen Ereignishandler für *DispatcherUnhandledException*:

```
...
    public partial class App : Application
    {
        private void Fehlerbehandlung(object sender, DispatcherUnhandledExceptionEventArgs e)
        {
            MessageBox.Show("Mal wieder ein Fehler: " + e.Exception.Message);
            e.Handled = true;
        }
    }
...
```

XAML

In der Datei *App.xaml* fügen Sie das Attribut *DispatcherUnhandledException* hinzu, um Ereignis und Ereignismethode miteinander zu verknüpfen:

```
<Application x:Class="WpfApplication2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
    DispatcherUnhandledException="Fehlerbehandlung" >
...
```

Tritt jetzt in Ihrem Programm ein unbehandelter Fehler auf, wird obige Ereignisprozedur ausgeführt und das Programm anschließend fortgesetzt.

12.5 Die Window-Klasse

Die eigentliche "Spielwiese" des Programmierers ist sicherlich das einzelne Fenster innerhalb der Anwendung. In WPF-Anwendungen handelt es sich dabei um Instanzen der Klasse *Window*, dem Pendant zu *Form* von Windows Forms-Anwendungen.

Bevor wir jetzt der Versuchung erliegen, alle Eigenschaften, Methoden und Ereignisse detailliert aufzulisten, wollen wir uns lieber einige spezifische Aufgabenstellungen herauspicken, vieles kennen Sie ja bereits von den Windows Forms.

12.5.1 Position und Größe festlegen

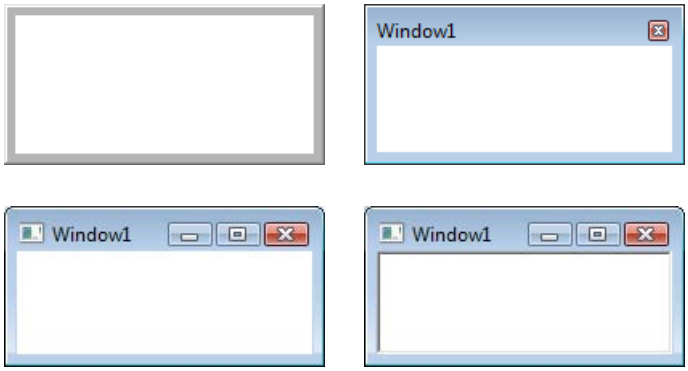
Haben Sie per Visual Studio ein neues Window erzeugt, sind bereits die Attribute für Breite und Höhe vorhanden (*Width, Height*) und müssen nur noch angepasst werden.

Die Position des Fensters können Sie zum einen mit dem Attribut *WindowStartupLocation* festlegen (*CenterOwner, CenterScreen, Manual*), zum anderen können Sie *Top* oder *Left* für die Ausrichtung verwenden.

Für die Position in der Liste der angezeigten Windows ist die Eigenschaft *Topmost* verantwortlich.

12.5.2 Rahmen und Beschriftung

Die Art des Rahmens beeinflussen Sie mit dem Attribut *WindowStyle (None, ToolWindow, SingleBorderWindow, ThreeDBorderWindow)*:



Die Beschriftung des Fensters lässt sich mit dem Attribut *Title* festlegen:

```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200">
```

12.5.3 Das Fenster-Icon ändern

Über das Attribut *Icon* lässt sich dem *Window* auch ein neues Icon (linke obere Ecke bzw. Taskleistenansicht) zuordnen.

BEISPIEL 12.45: Das Fenster-Icon festlegen

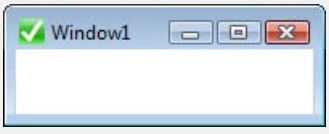
XAML

Ziehen Sie per Drag&Drop eine Icon-Datei in Ihr Projekt und legen Sie die *Icon*-Eigenschaft des *Window* auf deren Namen fest:

```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200" Icon="Sign.ico">
...
```

Ergebnis

Öffnen Sie jetzt das Fenster, wird das neue Icon angezeigt:



HINWEIS: Es ist besser, wenn Sie derartige Ressourcen in einem Unterverzeichnis Ihres Projekts speichern, so geht der Überblick nicht verloren. In diesem Fall müssen Sie für *Icon* auch den relativen Pfad angeben (z.B. "*\Images\Sign.ico*").

12.5.4 Anzeige weiterer Fenster

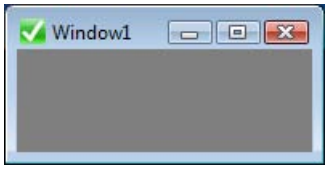
WPF-Windows zeigen Sie, wie auch die Windows Forms, nach dem Instanziieren mit den Methoden *Show* oder *ShowDialog* (modal) an.

BEISPIEL 12.46: Aufruf eines zweiten Fensters

C#
```
...
        private void button1_Click(object sender, RoutedEventArgs e)
        {
            Window2 w2 = new Window2();
            w2.ShowDialog();
...
        }
```

12.5.5 Transparenz

Legen Sie die Eigenschaft *Opacity* auf den gewünschten Wert der Transparenz (0 ... 1) fest, werden Sie sicher zunächst enttäuscht sein, denn es wird lediglich eine Graustufe angezeigt:



Der Grund: dieses Attribut wird nur im Zusammenhang mit dem Attribut *AllowsTransparency* berücksichtigt.

HINWEIS: Zusätzlich muss der Rahmentyp (*WindowStyle)* auf *None* festgelegt sein.

BEISPIEL 12.47: Teiltransparentes Window

XAML
```
<Window x:Class="WpfApplication3.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="200" Top="10" Left="10"
    Opacity="0.7" AllowsTransparency="True" WindowStyle="None">
```

BEISPIEL 12.47: Teiltransparentes Window

Ergebnis

Das erzeugte Fenster über dem Desktop:

12.5.6 Abstand zum Inhalt festlegen

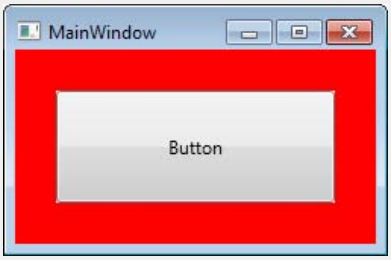
Über die Eigenschaft *BorderThickness* legen Sie fest, wie groß der Rahmen um den Clientbereich des Formulars ist. Die Rahmenfarbe legen Sie mit *BorderBrush* fest.

BEISPIEL 12.48: Verwendung von *BorderThickness*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded"
        BorderThickness="27"  BorderBrush="Red">
    <Grid>
        <Button Content="Button" Name="button1"  />
    </Grid>
</Window>
```

Ergebnis



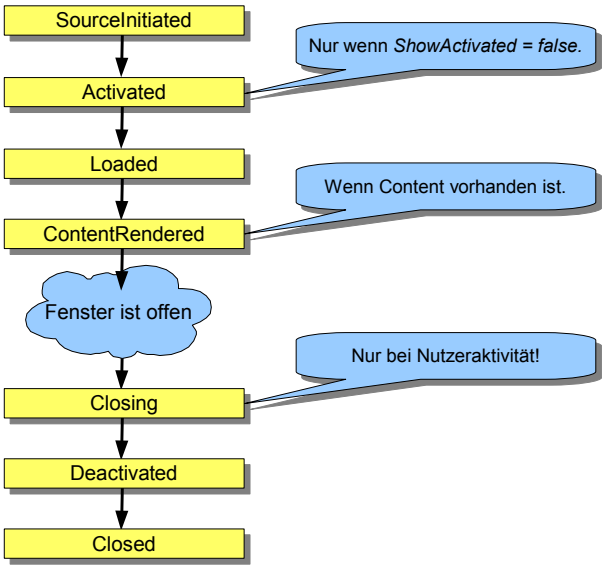
12.5.7 Fenster ohne Fokus anzeigen

Nicht in jedem Fall möchten Sie mit dem Einblenden eines neuen Fensters diesem auch den Eingabefokus zuweisen. Hier sei nur an Tool-Windows oder andere Statusfenster erinnert, die ohne Useraktivität auskommen.

Bevor Sie sich jetzt in endlose Programmierorgien stürzen, sollten Sie sich besser mit der *Window*-Eigenschaft *ShowActivated* vertraut machen. Setzen Sie diese auf *true*, wird das Fenster zwar geöffnet, aber es erhält nicht den Eingabefokus. Sie erkennen dies an der Rahmenfarbe, nicht aktivierte Fenster sind etwas heller.

12.5.8 Ereignisfolge bei Fenstern

Wie auch bei den Windows Forms möchten Sie als Programmierer auf diverse Ereignisse im "Leben" eines Windows reagieren können. Die WPF-Entwickler haben auch hier nicht gespart, für fast jede denkbare Situation steht ein Ereignis zur Verfügung:



HINWEIS: Weitere Informationen zum Ereignismodell von WPF finden Sie im Abschnitt 14.3 ab Seite 744.

12.5.9 Ein paar Worte zur Schriftdarstellung

Nachdem die ersten WPF-Versionen noch mit einer rudimentären Textausgabe aufgewartet haben[1], hatten die Entwickler wohl ein Einsehen und spendierten ab der Version 4.0 eine verbesserte Textausgabe, die vor allem bei kleinen Schriften nicht gleich zu Kopfschmerzen führt.

Ursache des Übels war der Algorithmus für das Rendern der Schriftarten. Dieser war zwar aus Programmierersicht akkurat (geräteneutral), auf TFT-Bildschirmen mit ihrer beschränkten Auflösung führte diese Darstellung jedoch zu fast unlesbaren Ergebnissen.

[1] Für einige Entwickler war die schlechte und verschwommene Textausgabe der alten Version **das** Killerargument gegen WPF.

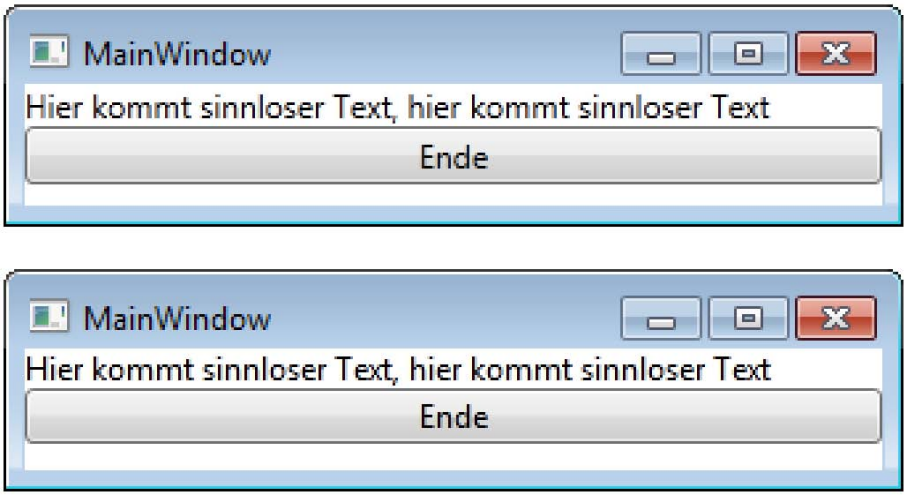
Über die neue Eigenschaft *TextOptions* bietet sich nun die Möglichkeit, die Darstellung von Schriftarten in Ihren Formularen zu optimieren. Zwei Optionen sind in diesem Zusammenhang von Bedeutung:

- *TextOptions.TextFormattingMode* und
- *TextOptions.TextRenderingMode*

HINWEIS: Beide Optionen können Sie für das gesamte Formular (wie im Weiteren beschrieben) oder auch nur für einzelne Controls zuweisen.

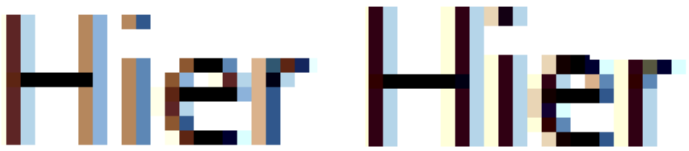
TextOptions.TextFormattingMode

Bevor wir lange um den heißen Brei herumreden, hier ein Beispiel für die Auswirkung von *TextOptions.TextFormattingMode* auf die Darstellung (Abbildung vergrößert):



Die obere Abbildung zeigt die Standarddarstellung von Schriften ("Ideal"), bei der unteren ist die Eigenschaft *Options.TextFormattingMode* auf "Display" gesetzt.

Beachten Sie bei der Darstellung insbesondere die senkrechten Linien (bei "H", "i" etc.). In der unteren Abbildung werden die teils grausigen Antialiasing-Artefakte drastisch entschärft, dies allerdings zu Lasten der Genauigkeit (siehe folgende Vergrößerung: links "Ideal", rechts "Display"):



Das folgende Beispiel zeigt, wie Sie die Option für das gesamte Formular setzen:

BEISPIEL 12.49: *TextOptions.TextFormattingMode* setzen

XAML
```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="84" Width="340" Loaded="Window_Loaded"
        TextOptions.TextFormattingMode="Display">
    <StackPanel>
        <TextBlock Text="Hier kommt sinnloser Text, hier kommt sinnloser Text"/>
        <Button>Ende</Button>
    </StackPanel>
```

Doch wann sollten Sie welche Option verwenden? Eine allgemeingültige Antwort lässt sich kaum geben, entweder Sie experimentieren etwas (unterschiedliche Bildschirme), oder Sie halten sich an folgende Pauschalaussagen:

- Verwenden Sie "Display" grundsätzlich bei kleinen Texten (< 15pt).
- Verwenden Sie die Standardeinstellung bzw. "Ideal" bei großen Texten, bei Transformationen, Zoom etc.

TextOptions.TextRenderingMode

Nach der bisherigen Kritik an der Textdarstellung wollten es die Entwickler wohl besonders gut machen und haben gleich noch eine weitere "Stellschraube" spendiert, die Sie für die Textoptimierung einsetzen können. Die Rede ist von *TextOptions.TextRenderingMode*, für das die drei Optionen *ClearType*, *Grayscale* und *Aliased* zur Verfügung stehen.

HINWEIS: Alle drei Optionen sind nur von Bedeutung, wenn die Eigenschaft *TextOptions.TextFormattingMode* auf *Display* gesetzt ist.

Ein Beispiel soll Ihnen die Unterschiede demonstrieren.

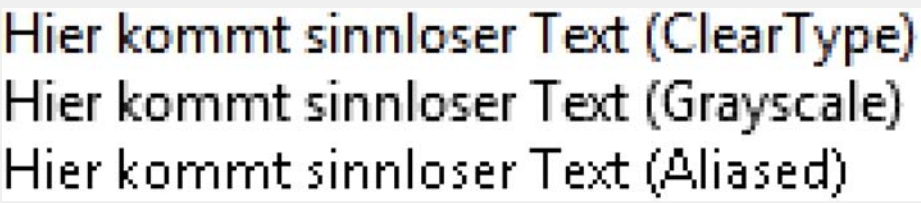
BEISPIEL 12.50: Verwendung von *TextOptions.TextRenderingMode*

XAML
```
Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="84" Width="340" Loaded="Window_Loaded"
        TextOptions.TextFormattingMode="Display">
    <StackPanel>
        <TextBlock>
            Hier kommt sinnloser Text (ClearType)
        </TextBlock>
        <TextBlock TextOptions.TextRenderingMode="Grayscale">
            Hier kommt sinnloser Text (Grayscale)
```

BEISPIEL 12.50: Verwendung von *TextOptions.TextRenderingMode*

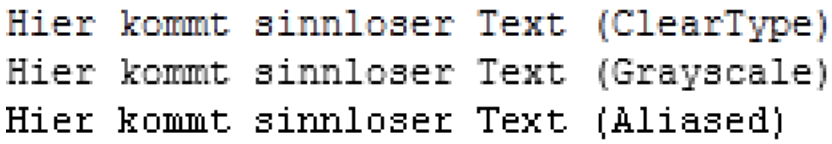
```
        </TextBlock>
        <TextBlock TextOptions.TextRenderingMode="Aliased">
            Hier kommt sinnloser Text (Aliased)
        </TextBlock>
    </StackPanel>
```

Ergebnis



Bei der Standard-Option "ClearType"wird Antialiasing mit diversen Farbabstufungen (Grau-, Brauntöne) realisiert, "Grayscale" nutzt hingegen lediglich Graustufen für die Kantenglättung. "Aliased" kommt ganz ohne Kantenglättung aus.

Wer jetzt auf die Idee kommt, "Cleartype" sei die Ideallösung für alle Einsatzfälle, sollte sich einmal die folgende Darstellung der Schriftart "Courier New" ansehen:



In diesem Fall dürfte der "klare" Sieger wohl eindeutig feststehen.

12.5.10 Ein paar Worte zur Darstellung von Controls

Nachdem wir uns bereits einige Seiten über die Schriftdarstellung ausgelassen haben, sollten wir auch noch kurz auf die Antialiasing-Effekte bei der Darstellung von Controls durch die Layoutengine eingehen. Auch hier gilt, dass die Auflösung von Bildschirmen begrenzt ist, und so bleibt der Layoutengine manchmal nichts anderes übrig, als eine berechnete Position/Breite durch einen Antialiasing-Effekt anzudeuten. So weit so gut, aber nicht jeder Anwender (insbesondere bei TFT-Bildschirmen) honoriert es, wenn Kanten nicht mehr klar, sondern verschwommen dargestellt werden, wie es das folgende Beispiel zeigt:

BEISPIEL 12.51: Antialiasing-Effekte bei Controls

XAML

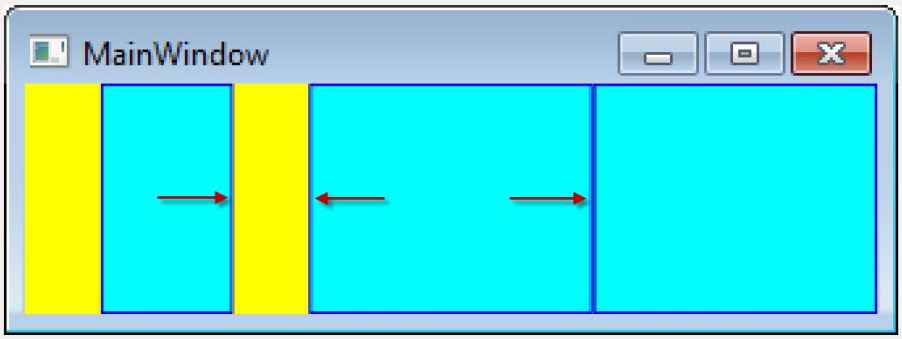
```
...
    <Grid Background="Yellow">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" /><ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
```

BEISPIEL 12.51: Antialiasing-Effekte bei Controls

```
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Width="50.5"/>
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Grid.Column="1"/>
        <Rectangle Stroke="Blue" StrokeThickness="1" Fill="Aqua" Grid.Column="2"/>
    </Grid>
...
```

Ergebnis

Beachten Sie die verschwommenen senkrechten Kanten der Rechtecke:

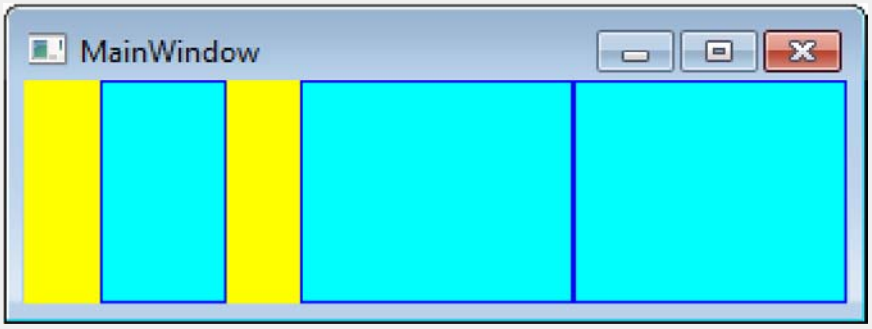


BEISPIEL 12.52: Ausschalten der Effekte mit *UseLayoutRounding*

XAML

```
...
    <Grid UseLayoutRounding="True" Background="Yellow">
...
```

Ergebnis



Wie Sie dem zweiten Beispiel entnehmen können, führt das Setzen von *UseLayoutRounding* auf *True* dazu, dass kritische Kanten so verschoben werden, dass sie mit den Bildschirmpixeln übereinstimmen. Dies geht zwar etwas zu Lasten der Genauigkeit, im Interesse einer klaren Darstellung können wir hier aber sicher "ein Auge zudrücken".

12.5.11 Wird mein Fenster komplett mit WPF gerendert?

Nein, Rahmen und Kopfzeile werden nach wie vor durch GDI-Anweisungen generiert, was auch erklärt, warum viele WPF-Features (Styles, Transparenz, Rotation ...) nicht auf das eigentliche Fenster angewendet werden können.

Damit haben wir uns mit den notwendigsten Grundkenntnissen über die WPF-Anwendung versorgt.

HINWEIS: In den weiteren Kapiteln wollen wir uns zunächst den wichtigsten Controls zuwenden (Kapitel 13), bevor wir zu speziellen Themen wie Ereignisbehandlung Styles etc. kommen (Kapitel 14). Das Spezialthema "Datenbindung finden Sie im Kapitel 15, Kapitel 16 widmet sich abschließend der Druckausgabe in WPF.

Kapitel 13

Übersicht WPF-Controls

Bei den WPF-Controls handelt es sich, ähnlich wie bei den Windows Forms-Steuerelementen, um ein recht komplexes Thema. Angesichts der erdrückenden Vielfalt unterschiedlichster Controls und deren Features können wir auch hier keinen Anspruch auf Vollständigkeit erheben.

13.1 Allgemeingültige Eigenschaften

Da die meisten Controls direkt oder indirekt von der Klasse *Control* (*System.Windows.Controls*) abgeleitet sind (und diese wiederum von *FrameworkElement* ...), verfügen sie auch über einige gemeinsame Eigenschaften, die wir an dieser Stelle kurz besprechen wollen.

Eigenschaft	Beschreibung
Background	die Füll-/Hintergrundfarbe, z.B. `<Button Background="Blue">`
BorderBrush	die Rahmenfarbe, z.B. `<Button BorderBrush="Blue">`
Cursor	der Mauszeiger für das Control, z.B. `<Button Cursor="Wait">`
FontFamily, FontSize, FontStyle, FontWeight	Schriftart, -größe, -schnitt (*Normal*, *Italic*, *Oblique*), -breite (*Light*, *Normal*, *UltraBold*), z.B. `<Button FontFamily="Arial" FontSize="12" FontStyle="Italic" FontWeight="UltraBold">`
Foreground	Vordergrundfarbe, z.B. `<Button Foreground="Coral">`
HorizontalAlignment, VerticalAlignment	Horizontale/vertikale Ausrichtung bezüglich des übergeordneten Elements, z.B. `<Button HorizontalAlignment="Left">`

Eigenschaft	Beschreibung
HorizontalContentAlignment, VerticalContentAlignment	Horizontale/vertikale Ausrichtung der Inhalte, z.B. `<Button HorizontalContentAlignment="Left">`
Height/Width	Höhe und Breite des Controls, siehe auch Seite 602
Margin	Außenabstände, siehe auch Seite 604
MaxHeight, MaxWidth, MinHeight, MinWidth	maximale bzw. minimale Abmessungen des Controls
Name	Name der Komponente, z.B. `<Button Name="button1">`
Opacity	Transparenz (ein Wert zwischen 0 und 1), z.B. `<Button Opacity="0.5">`
Padding	Innenabstände, siehe auch Seite 604
Parent	das übergeordnete Element
Resources	Verweis auf Ressourcen, dazu mehr ab Seite 739
Style	Verweis auf einen Style, dazu mehr ab Seite 754
TabIndex	Index in der Tabulatorreihenfolge
Tag	Frei definierbare Eigenschaft vom Typ *object*, hier können Sie eigene Informationen speichern
ToolTip	Die beliebten kleinen Fähnchen, die dem Endanwender "unnötige" Informationen aufdrängen
Visibility	Die Sichtbarkeit des Controls (*Collapsed*, *Hidden*, *Visible*), Achtung: Bei *Hidden* beansprucht das Control nach wie vor seinen Platz, ist nur nicht sichtbar. Mit *Collapsed* verschwindet auch die "Leerstelle".

Ergänzend wollen wir noch auf zwei Eigenschaften eingehen, die über einen anderen Namespace in den XAML-Code importiert werden, aber häufig benötigt werden:

Eigenschaft	Beschreibung
x:Name	Ermöglicht es, Objekten die nicht von *FrameworkElement* abgeleitet sind, einen Bezeichner zu geben, um zur Laufzeit auf dieses Objekt zuzugreifen.
x:FieldModifier	Mit dem Zugriffsmodifizierer steuern Sie die Sichtbarkeit des Objekts außerhalb der Klasse (*public* oder *internal*).

BEISPIEL 13.1: Der Unterschied zwischen *Hidden* (links) und *Collapsed* (rechts) für *Button2*

XAML

```
<StackPanel>
  <Button>Button1</Button>
  <Button Visibility="Collapsed">Button2</Button>
  <Button>Button3</Button>
</StackPanel>
```

Ergebnis

HINWEIS: Umsteiger werden sicher darauf hereinfallen: Die Höhe und Breite des Controls rufen Sie nicht über *Height* bzw. *Width,* sondern über *ActualHeight* und *ActualWidth* ab. Der Grund ist in der Verwendung der Layouts zu finden, d.h., die übergeordneten Elemente bestimmen die Abmessungen des Controls. Eine Vorgabe für Breite und Höhe wird jedoch nach wie vor über *Width* und *Height* realisiert.

13.2 Label

Dieses innovative Control kennen Sie sicher noch zur Genüge aus Ihrer "Vor"-WPF-Ära, der Verwendungszweck ist zum einen das Beschriften von Controls (z.B. *TextBox*), zum anderen lassen sich so zum Beispiel Access-Keys einer *TextBox* zuordnen. Die Zuordnung zwischen *Label* und Control erfolgt mit dem Attribut *Target*, wie es das folgende Beispiel zeigt.

BEISPIEL 13.2: Festlegen eines Access-Key (Alt+E) und Zuordnen der *TextBox1* (Ziel des Eingabefokus)

XAML

```
...
  <StackPanel Grid.Column="0" Grid.Row="0" >
    <Label Target="{Binding ElementName=textBox1}" Name="label1" >
    _Edit
    </Label>
    <TextBox Name="textBox1" />
  </StackPanel>
...
```

Ergebnis

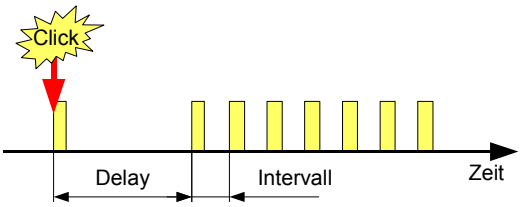
Die Laufzeitansicht (nach dem Drücken der Alt-Taste):

13.3 Button, RepeatButton, ToggleButton

Der Verwendungszweck von *Button, RepeatButton* und *ToggleButton* als Schaltfläche mit *Click*-Ereignis dürfte schnell klar sein. Doch es gibt einige Unterschiede, wie die folgende Tabelle zeigt:

Element	Beschreibung
Button	Ein einzelnes *Click*-Ereignis nach Betätigung.
RepeatButton	Mehrfache *Click*-Ereignisse bei Betätigung, die Frequenz wird mit der Eigenschaft *Intervall* (Millisekunden) bestimmt, der Abstand zwischen dem ersten Klick (resultiert aus dem Niederdrücken) und der Wiederholung kann mit *Delay* (Millisekunden) festgelegt werden (siehe folgende Abbildung).
ToggleButton	Einzelnes *Click*-Ereignis nach Betätigung, die Schaltfläche schaltet jedoch mit jedem Klick zwischen den Zuständen "gedrückt" und "nicht gedrückt" um. Zur Auswertung des Status stehen die beiden Ereignisse *Checked* und *UnChecked* sowie die Eigenschaft *IsChecked* zur Verfügung.

Das Verhalten des *RepeatButton*:



BEISPIEL 13.3: Einige Schaltflächendefinitionen

XAML

```
<StackPanel>
  <Button Margin="6" Click="Button_Click">Button</Button>
  <RepeatButton Delay="2000" Interval="500" Click="RepeatButton_Click">RepeatButton
  </RepeatButton>
  <ToggleButton  Name="tb" Click="RepeatButton_Click">ToggleButton</ToggleButton>
</StackPanel>
```

13.3.1 Schaltflächen für modale Dialoge

Eine Standardaufgabe für den Programmierer ist häufig das Erstellen von modalen Dialogen, deren Ergebnis (OK, Abbruch) im übergeordneten Fenster ausgewertet werden muss. Wird das Window mit *Alt+F4*, der Schließen-Schaltfläche oder per Systemmenü geschlossen, wird die *DialogResult*-Eigenschaft des betreffenden Windows auf *False* gesetzt. Den Wert *True* müssen Sie z.B. über die OK-Schaltfläche explizit setzen.

Die standardmäßige Zuordnung der beiden Tasten (OK/Abbruch) können Sie mit den Eigenschaften *IsDefault* (Verknüpfung mit Enter-Taste) und *IsCancel* (Verknüpfung mit *ESC*-Taste) realisieren. *IsCancel=True* führt zum automatischen Schließen des Fensters.

BEISPIEL 13.4: Aufruf eines modalen Dialogs und Auswertung von OK/Abbruch

XAML

Das modale Window:

```
<Window x:Class="WpfApplication3.Window2"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window2" Height="300" Width="300">
    <Grid>
     <Grid.RowDefinitions>
       <RowDefinition Height="216*" />
       <RowDefinition Height="46*" />
     </Grid.RowDefinitions>
     <Button IsDefault="True" Grid.Row="1" Margin="89,6,104,8" Name="button1"
      Click="button1_Click">Ok</Button>
     <Button IsCancel="True" Grid.Row="1" HorizontalAlignment="Right" Margin="0,6,14,10"
      Name="button2" Width="79">Abbruch</Button>
    </Grid>
</Window>
```

C#

Die Ereignisprozedur für den OK-Button:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    this.DialogResult = true;
    this.Close();
}
```

Der Aufruf des modalen Window und Auswertung:

```
Window2 w2 = new Window2();
w2.ShowDialog();
if ((bool)w2.DialogResult)
    MessageBox.Show("OK");
else
    MessageBox.Show("Abbruch");
```

13.3.2 Schaltflächen mit Grafik

Wer jetzt verzweifelt nach einer Image-Eigenschaft Ausschau hält, dürfte enttäuscht sein. Keine der Schaltflächen verfügt darüber, was jedoch kein Problem ist, da Sie in WPF-Anwendungen den Content des Controls selbst bestimmen können (*ContentControl*).

Doch wie müssen wir vorgehen? Ausschlaggebend für die Zusammenstellung unserer Schaltfläche ist das zukünftige Layout (Grafik oben, Texte unten oder anders herum). Mit den im Abschnitt 12.3 vorgestellten Layout-Controls können Sie zunächst ein geeignetes Layout für den Inhalt der Schaltfläche entwerfen und dann die eigentlichen Image- und Text-Elemente[1] anordnen.

[1] Statt einer Grafik könnten Sie auch gleich ein komplettes Video in der Schaltfläche anzeigen. Ob dies sinnvoll ist, ist eine andere Frage.

BEISPIEL 13.5: Schaltfläche mit Text und Grafik

C#

```
<Button Click="Button_Click">
```

Ein *StackPanel* bestimmt das innere Layout der Schaltfläche:

```
    <StackPanel Orientation="Horizontal" Margin="10">
```

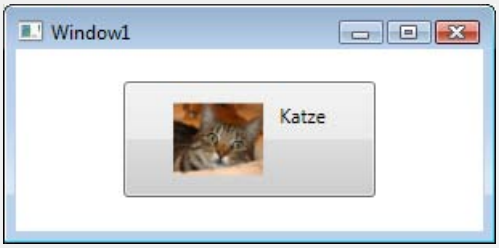
Die Grafik festlegen (ziehen Sie vorher einfach eine Grafik per Drag&Drop in Ihr Projekt):

```
      <Image Source="Annie in the Sink.jpg" Width="56" Height="46" Margin="0,0,10,0"/>
```

Der Beschriftungstext (hier könnten Sie auch mit Textformatierungen arbeiten):

```
      <TextBlock VerticalAlignment="Center">Katze</TextBlock>
    </StackPanel>
  </Button>
```

Ergebnis



HINWEIS: Statt dieser Lösung können Sie natürlich auch die *Button*-Klasse ableiten, oder Sie nutzen die Möglichkeit eines UserControls. Beide Varianten sollen aber nicht im Mittelpunkt dieses Abschnitts stehen, siehe dazu unser [Visual C# 2012 Kochbuch].

13.4 TextBox, PasswordBox

Beide Controls dürften Ihnen als wichtige Möglichkeiten für die Texteingabe sicher bekannt sein. An dieser Stelle wollen wir deshalb nur auf einige Eigenschaften zur Konfiguration dieser Eingabefelder eingehen.

13.4.1 TextBox

Die wichtigsten Eigenschaften auf einen Blick:

Eigenschaft	Beschreibung
Text	Der Inhalt der *TextBox*.
TextWrapping	Einzeilige (*NoWrap*) oder mehrzeilige (*Wrap, WrapWithOverFlow*) Darstellung. *Wrap* bricht bei jedem Zeichen um, *WrapWithOverFlow* nur an Leerzeichen.

Eigenschaft	Beschreibung
AcceptReturn	Wenn *True,* erzeugt die *Enter*-Taste eine neue Zeile, andernfalls wird Enter ignoriert.
IsReadOnly	Schreibschutz ja/nein.
VerticalScrollBarVisibility, HorizontalScrollBarVisibility	Sichtbarkeit der Scrollbars vorgeben. Mögliche Werte: *Auto, Disabled, Hidden, Visible*
MaxHeight, MaxWidth, MinHeight, MinWidht	Maximale/minimale Abmessungen der *TextBox* vorgeben.
MaxLines, MinLines	Minimale/maximale Anzahl von angezeigten Zeilen in der *TextBox*. Nur sinnvoll mit *TextWrapping*.
MaxLength	Die maximale Zeichenzahl.
CaretIndex	Position des Eingabekursors.
SpellCheck.IsEnabled	Ein-/Ausschalten der integrierten Rechtschreibkorrektur (siehe Beispiel).

BEISPIEL 13.6: Setzen der *Text*-Eigenschaft per XAML

XAML

```
<TextBox Text="Meine Vorgabe">
</TextBox>
```

oder

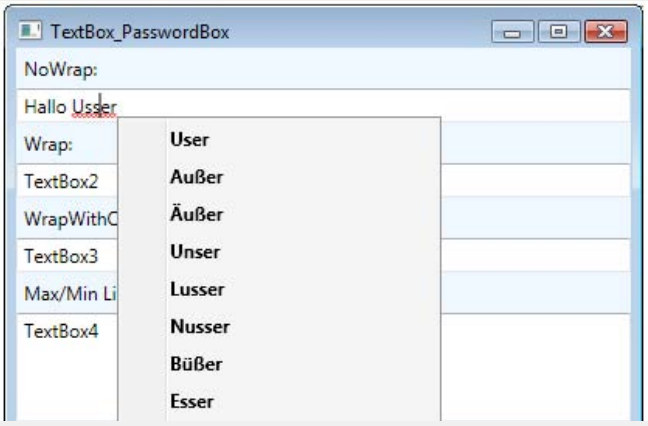
```
<TextBox>
  Meine Vorgabe
</TextBox>
```

BEISPIEL 13.7: Verwendung der Rechtschreibkorrektur

XAML

```
<TextBox SpellCheck.IsEnabled="True" Name="TextBox1" >TextBox1</TextBox>
```

Ergebnis



Zur Laufzeit wird jetzt, je nach aktueller Landeseinstellung, eine automatische Rechtschreibprüfung durchgeführt, Fehler werden rot unterstrichen und Alternativen per Kontextmenü angezeigt.

Und da Deutsch eine Sprache mit andauernden behördlichen Änderungen ist, findet sich auch eine Möglichkeit, die aktuelle Rechtschreibreform zu berücksichtigen. Verwenden Sie einfach die Eigenschaft *SpellingReform* und setzen Sie einen der folgenden Werte (*Prerefrom*, *Postrefrom*, *PreAndPostrefrom*)[1].

Wahrscheinlich haben Sie dieses Feature bis jetzt noch gar nicht vermisst: seit WPF 4 können Sie sowohl für den Eingabekursor als auch für die Kursorauswahl einen extra *Brush* verwenden, was Ihren gestalterischen Fähigkeiten natürlich jede Menge Raum gibt.

BEISPIEL 13.8: Texteingabe- und Auswahlkursor ändern

XAML

Hier zunächst der geänderte Eingabekursor:

```
<TextBox FontSize="20" CaretBrush="Green">
```

Und hier ein etwas komplexerer Auswahlkursor:

```
    <TextBox.SelectionBrush>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="Transparent" Offset="-.2" />
            <GradientStop Color="Red" />
            <GradientStop Color="Yellow" Offset=".9" />
            <GradientStop Color="Transparent" Offset="1.2" />
        </LinearGradientBrush>
    </TextBox.SelectionBrush>
    Hallo User
</TextBox>
```

Ergebnis

13.4.2 PasswordBox

Der Sinn dieses Controls ist die verdeckte Eingabe von Zeichenketten (Passwörtern). Für Sie als Programmierer sind die beiden Eigenschaften

- *PasswordChar* (das Maskierungszeichen) und
- *Password* (die eingegebene Zeichenkette)

interessant.

[1] In .NET 5.x findet sich dann sicher auch ein *PostPostNextPreReform*

BEISPIEL 13.9: Verwendung *PasswordBox*

XAML

```
<PasswordBox Name="Pwd1" MaxLength="6" Password="geheim" PasswordChar="?"
             KeyUp="PasswordBox_KeyUp"/>
```

C#

Die Ereignis-Routine:

```
private void PasswordBox_KeyUp(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
        MessageBox.Show(Pwd1.Password);
}
```

13.5 CheckBox

Geht es um die Auswahl bzw. Darstellung boolescher Werte (*True*/*False),* bietet sich neben dem *ToggleButton* die gute alte *CheckBox* an.

Die Beschriftung des Controls bestimmen Sie über die *Content*-Eigenschaft, hier ist stattdessen z.B. auch ein *Image* möglich, was optisch sicher viel hergibt (siehe Beispiel am Abschnittsende).

Den aktuellen Status können Sie über die Eigenschaft *IsChecked* bestimmen.

HINWEIS: Achtung: Hier sind drei Zustände (*Null, True, False*) möglich, wenn Sie die Eigenschaft *IsThreeState* auf *True* setzen (unbestimmte Angabe).

Die Auswertung einer Änderung kann in den Ereignissen *Checked* bzw. *UnChecked* erfolgen, besser ist jedoch *Click*, da Sie hier beide Zustände auswerten können.

BEISPIEL 13.10: Verwendung der *CheckBox*

XAML

```
<StackPanel Margin="5">
```

Variante 1:

```
  <CheckBox Checked="CheckBox_Checked" Unchecked="CheckBox_Unchecked"
Click="CheckBox_Click"
            Name="checkBox1" Foreground="Blue">Ich kann lesen</CheckBox>
```

Variante 2:

```
  <CheckBox Name="checkBox2" Foreground="Green">Ich kann schreiben</CheckBox>
```

Hier mit drei Zuständen:

```
  <CheckBox Name="checkBox3" IsThreeState="True" IsChecked="">Schwanger</CheckBox>
```

CheckBox mit Grafik statt Text:

```
  <CheckBox Name="checkBox4" Foreground="Green">
    <Image Source="Images\rudi.gif" Width="50" Height="50"/>
```

BEISPIEL 13.10: Verwendung der *CheckBox*

```
</CheckBox>
```

Natürlich könnte es auch jedes andere Control sein, das Sie als Content der *CheckBox* festlegen, aber einen Sinn sollte es schon machen.

```
</StackPanel>
```

C#

Die Ereignisroutinen:

```
private void CheckBox_Checked(object sender, RoutedEventArgs e)
{
    MessageBox.Show("True");
}

private void CheckBox_Unchecked(object sender, RoutedEventArgs e)
{
    MessageBox.Show("False");
}
```

Nur *True/False* auswerten:

```
private void CheckBox_Click(object sender, RoutedEventArgs e)
{
    if ((bool)checkBox1.IsChecked)
        MessageBox.Show("True");
    else
        MessageBox.Show("False");
}
```

Unbestimmten Zustand auswerten:

```
private void checkBox3_Click(object sender, RoutedEventArgs e)
{
    if (checkBox3.IsChecked == null) MessageBox.Show("Etwas Schwanger!");
}
```

Ergebnis

Beachten Sie die obige Option "Schwanger": angezeigt wird der unbestimmte Zustand, d h. in diesem Fall "etwas schwanger".

13.6 RadioButton

Möchten Sie mehr als nur eine *True*/*False*-Auswahl anbieten (siehe *CheckBox),* können Sie eine Reihe von Radiobuttons verwenden, bei der jeweils nur ein *RadioButton* markiert sein kann.

Um die einzelnen Radiobuttons miteinander zu verbinden, steht Ihnen die Eigenschaft *GroupName* zur Verfügung. Alle Controls mit übereinstimmenden Namen werden zu einer Gruppe zusammengefasst, nur ein Element der Gruppe kann markiert sein (*IsChecked*).

BEISPIEL 13.11: Verwendung *RadioButton*

XAML

```
<StackPanel Margin="5">
  <Label Content="----------------RadioButton ----------------------" Margin="4,10"/>
```

Die einfachste Variante:

```
  <RadioButton Name="RB1">Option1</RadioButton>
```

Mit gesetzter *IsChecked*-Option:

```
  <RadioButton Name="RB2" IsChecked="True">Option2</RadioButton>
```

Als Content ist auch ein Image oder jedes andere Control zulässig:

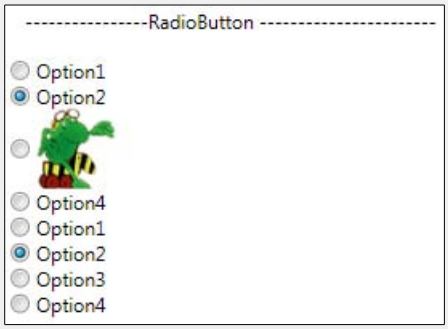
```
  <RadioButton Name="RB3">
      <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </RadioButton>
  <RadioButton Name="RB4">Option4</RadioButton>
```

Und hier beginnt eine neue Gruppe, da anderer Parent:

```
  <StackPanel>
    <RadioButton Name="RB5">Option1</RadioButton>
    <RadioButton Name="RB6" IsChecked="True">Option2</RadioButton>
    <RadioButton Name="RB7">Option3</RadioButton>
    <RadioButton Name="RB8">Option4</RadioButton>
  </StackPanel>
</StackPanel>
```

Ergebnis

Wie Sie sehen können, sind bei diesem Beispiel zwei *RadioButton*s markiert:



HINWEIS: Geben Sie keinen *GroupName* an, werden alle *RadioButton*s mit gemeinsamem Parent (z.B. *StackPanel*) zu einer Gruppe zusammengefasst.

13.7 ListBox, ComboBox

Für die Auswahl aus Listen mit mehreren Einträgen bieten sich statt der *RadioButton*s besser eine *ListBox* oder eine *ComboBox* an. Erstere gestattet die gleichzeitige Anzeige mehrerer Werte, die *ComboBox* stellt nur einen Wert aus der Liste dar.

13.7.1 ListBox

Jede *ListBox* kann mehrere Einträge, d h. *ListBoxItems*, enthalten, von denen wiederum ein oder mehr Einträge ausgewählt sein können.

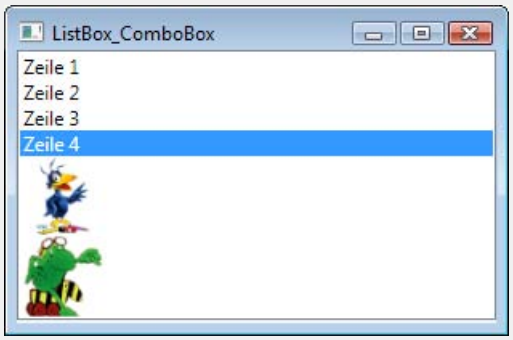
Sollen mehrere Einträge ausgewählt werden können, setzen Sie *SelectionMode* auf *Multiple (*Auswahl durch einfachen Klick) oder *Extended* (Auswahl durch Klick + Shift-Taste).

BEISPIEL 13.12: Definition einer *ListBox* mit 6 Einträgen inklusive zweier Grafiken

XAML

```
<ListBox SelectionMode="Extended">
  <ListBoxItem IsSelected="True">Zeile 1</ListBoxItem>
  <ListBoxItem>Zeile 2</ListBoxItem>
  <ListBoxItem>Zeile 3</ListBoxItem>
  <ListBoxItem>Zeile 4</ListBoxItem>
  <ListBoxItem>
     <Image Source="Images\rudi.gif" Width="50" Height="50"/>
  </ListBoxItem>
  <ListBoxItem>
     <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </ListBoxItem>
</ListBox>
```

Ergebnis



Alternativ können Sie die Beschriftung auch mit einer etwas kürzeren Schreibweise zuweisen:

```
<ListBoxItem Content="Zeile 4"/>
```

HINWEIS: Im Gegensatz zu den Windows Forms-Listboxen können Sie jeden Eintrag individuell (Farbe, Hintergrundfarbe, Schrift etc.) formatieren. Ob dies sinnvoll und der Übersicht dienlich ist, sei dahingestellt.

Wem diese Möglichkeiten nicht ausreichen, der kann auch andere Controls der *ListBox* als Einträge hinzufügen.

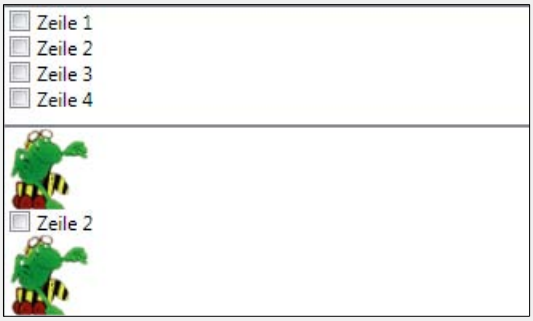
BEISPIEL 13.13: Controls in der *ListBox* (eine einfache "CheckListBox"):

XAML

```
<ListBox SelectionMode="Multiple" Height="75">
  <CheckBox Name="Item1">Zeile 1</CheckBox>
  <CheckBox Name="Item2">Zeile 2</CheckBox>
  <CheckBox Name="Item3">Zeile 3</CheckBox>
  <CheckBox Name="Item4">Zeile 4</CheckBox>
</ListBox>
<ListBox >
  <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  <CheckBox Name="Item7">Zeile 2</CheckBox>
  <Image Source="Images\frosch.gif" Width="50" Height="50"/>
</ListBox>
```

Ergebnis

Die beiden erzeugten *ListBox*en:



Doch wie können Sie die aktuelle Markierung bzw. die Auswahl ermitteln? Die *ListBox* stellt zu diesem Zweck eine Reihe von Eigenschaften zur Verfügung, von denen *SelectedIndex* (die erste markierte Zeile, nullbasiert) und *SelectedItems* (Collection von *ListBoxItems*) am nützlichsten sind.

HINWEIS: Prüfen Sie vor der Verwendung gegebenenfalls, ob überhaupt ein Eintrag markiert wurde. In diesem Fall muss *SelectedIndex* einen Wert ungleich -1 aufweisen.

BEISPIEL 13.14: *ListBox* mit Einträgen füllen und später die markierten abfragen

XAML

XAML-Code zum Füllen der *ListBox:*

```
<ListBox Name="ListBox1" SelectionMode="Extended">
  <ListBoxItem>Zeile 1</ListBoxItem>
  <ListBoxItem>Zeile 2</ListBoxItem>
  <ListBoxItem>Zeile 3</ListBoxItem>
  <ListBoxItem>Zeile 4</ListBoxItem>
```

Eintrag mit Grafik:

```
  <ListBoxItem>
    <Image Source="Images\rudi.gif" Width="50" Height="50"/>
  </ListBoxItem>
  <ListBoxItem>
    <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </ListBoxItem>
</ListBox>
```

C#

Auswerten per Ereigniscode:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    if (ListBox1.SelectedIndex != -1)
    {
        MessageBox.Show(ListBox1.SelectedValue.ToString());
        MessageBox.Show("Es ist mindestens ein Eintrag markiert!");
        MessageBox.Show("Zeile " + ListBox1.SelectedIndex.ToString() +
                        " ist markiert!");
        foreach (ListBoxItem item in ListBox1.SelectedItems)
            MessageBox.Show(item.Content.ToString());
    }
}
```

HINWEIS: Für normale Textzeilen wird obiger Code die sichtbare Beschriftung zurückgeben, für markierte Grafik-Einträge kommt der Text "System.Windows.Controls.Image".

BEISPIEL 13.15: ListBoxeintrag zur Entwurfszeit markieren

XAML

```
...
  <ListBoxItem IsSelected="True">Four</ListBoxItem>
...
```

Natürlich müssen Sie nicht unbedingt auf XAML zurückgreifen, wenn Sie eine *ListBox* füllen wollen. Dies funktioniert genauso gut auch per Code.

BEISPIEL 13.16: *ListBox* zur Laufzeit füllen

C#

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    ListBox2.Items.Clear();             // Alles löschen
    CheckBox cb = new CheckBox();       // zur Probe mal eine CheckBox einfügen
    cb.Content = "Ein Test";
    ListBox2.Items.Add(cb);
    for (int i = 1; i < 50; i++)        // und jetzt noch fünfzig Zeilen Text ...
        ListBox2.Items.Add("Zeile" + i.ToString());
}
```

Ergebnis

HINWEIS: Möchten Sie die Einträge einzeln formatieren, verwenden Sie *ListBoxItems* beim Aufruf von *Add*.

13.7.2 ComboBox

Das Handling aus Sicht des Programmierers entspricht weitgehend der *ListBox*, mit dem wesentlichen Unterschied, dass der Anwender nur ein Element der Auswahlliste markieren kann. Dieses wird nachfolgend in der *ComboBox* angezeigt (Eigenschaft *Text)*.

Neben der reinen Auswahl von vorgegebenen Werten können Sie auch neue Einträge zulassen. Dazu müssen Sie die Eigenschaft *IsEditable* auf *True* setzen. In diesem Fall erscheint eine Textbox, die der Nutzer nach Wunsch füllen kann.

BEISPIEL 13.17: Einfache *ComboBox*

XAML

```
<ComboBox Name="ComboBox1"  SelectionChanged="ComboBox1_SelectionChanged">
  <ComboBoxItem>Zeile 1</ComboBoxItem>
  <ComboBoxItem>Zeile 2</ComboBoxItem>
  <ComboBoxItem>Zeile 3</ComboBoxItem>
  <ComboBoxItem>Zeile 4</ComboBoxItem>
  <ComboBoxItem>Zeile 5</ComboBoxItem>
  <ComboBoxItem>
    <Image Source="Images\rudi.gif" Width="50" Height="50"/>
  </ComboBoxItem>
  <ComboBoxItem>
    <Image Source="Images\frosch.gif" Width="50" Height="50"/>
  </ComboBoxItem>
</ComboBox>
```

BEISPIEL 13.17: Einfache *ComboBox*

C#

Ereignisauswertung:

```
private void ComboBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    MessageBox.Show("Markierte Zeile: " + ComboBox1.SelectedIndex.ToString());
}
```

Ergebnis

HINWEIS: In obiger Ereignisprozedur ist die *Text*-Eigenschaft noch **nicht** auf den neuen Wert gesetzt, dies erfolgt (aus unerfindlichen Gründen) erst zu einem späteren Zeitpunkt.

Die *ComboBox* in Aktion:

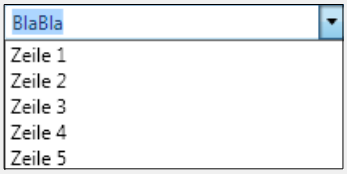
BEISPIEL 13.18: *ComboBox* mit Texteingabe

XAML

```
<ComboBox Name="ComboBox2" IsEditable="True">
  <ComboBoxItem>Zeile 1</ComboBoxItem>
  <ComboBoxItem>Zeile 2</ComboBoxItem>
  <ComboBoxItem>Zeile 3</ComboBoxItem>
  <ComboBoxItem>Zeile 4</ComboBoxItem>
  <ComboBoxItem>Zeile 5</ComboBoxItem>
</ComboBox>
```

Ergebnis

Die Texteingabe zur Laufzeit:



HINWEIS: Auf das Füllen per Datenbindung gehen wir erst in Kapitel 15 (ab Seite 787) ein.

13.7.3 Den Content formatieren

Gut und schön werden Sie sicher sagen, vieles davon kann auch Windows Forms. Doch da täuschen Sie sich vermutlich etwas. Die einzelnen *ComboBoxItem*- bzw. *ListBoxItem*-Einträge verfügen nicht nur über die *Content*-Eigenschaft sondern bieten darüber hinaus auch noch reichlich Optionen für das gezielte Formatieren des Inhalts an.

Statt endloser Aufzählungen soll ein Beispiel einige Anregungen bieten:

BEISPIEL 13.19: Formatieren von *ComboBox*-Einträgen

XAML

```
<ComboBox Width="150" FontSize="16" BorderThickness="2" BorderBrush="Blue">
    <ComboBoxItem HorizontalAlignment="Right" Foreground="Green">Zeile 1
    </ComboBoxItem>
    <ComboBoxItem FontWeight="Bold" FontFamily="Courier New">Zeile 2</ComboBoxItem>
    <ComboBoxItem Width="50" Background="Yellow">Zeile 3</ComboBoxItem>
    <ComboBoxItem FontSize="8">Zeile 4</ComboBoxItem>
    <ComboBoxItem Background="Gray" Content="Zeile 5" />
</ComboBox>
```

Ergebnis

Unter bestimmten Umständen können Sie statt der ausführlichen Syntax auch eine verkürzte Form verwenden, allerdings müssen Sie in diesem Fall einen zusätzlichen Namespace einbinden und die Einträge typisieren:

BEISPIEL 13.20: String-Formatierung in einer *ComboBox*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib" Height="127" Width="340">
    <StackPanel>
        <ComboBox Width="150" HorizontalContentAlignment="Center" ItemStringFormat="0.00 €">
            <sys:String>Kein Wert</sys:String>
            <sys:Decimal>0.1</sys:Decimal>
            <sys:Decimal>1</sys:Decimal>
            <sys:Decimal>1.1</sys:Decimal>
            <sys:Decimal>1.11</sys:Decimal>
            <sys:Decimal>1.111</sys:Decimal>
        </ComboBox>
    </StackPanel></Window>
```

BEISPIEL 13.20: String-Formatierung in einer *ComboBox*

Ergebnis

13.8 Image

Ganz nebenbei haben wir in den vorhergehenden Beispielen bereits einen Blick auf das *Image*-Control geworfen. Wie Sie sicher schon festgestellt haben, können Sie damit Grafiken auf einfache Weise sowohl in einem *Window*, als auch in ListBoxen, ComboBoxen, Buttons etc. anzeigen.

Zur Verfügung stehen neben den altbekannten Formaten BMP, GIF, ICO JPG, PNG, TIFF auch das WDP-Format[1]. Leider werden sowohl das WMF als auch das EMF-Format nicht unterstützt, hier hilft nur die Verwendung von Windows Forms.

13.8.1 Grafik per XAML zuweisen

Zugewiesen wird die Grafik über das *Source*-Attribut. Die genaue Form der Adressierung von Programmressourcen besprechen wir ausführlich erst in Abschnitt 14.2 (ab Seite 739), an dieser Stelle soll uns ein Beispiel genügen.

BEISPIEL 13.21: Grafik aus Ressource laden

XAML

Erstellen Sie zunächst per Kontextmenü einen neuen Ordner[2] (*Images)* für Ihr WPF-Projekt. Fügen Sie nachfolgend per Drag&Drop eine Grafik (z.B. *Frosch.gif*) in dieses Verzeichnis ein.

Der XAML-Code:

```
<Window x:Class="BSP_Controls.Image_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Image_Bsp" Height="300" Width="300">
       <Image Source="Images\frosch.gif"/>
</Window>
```

[1] Windows Media Photo-Format

[2] Dies ist nicht zwingend erforderlich, verbessert aber die Übersicht im Projekt.

13.8.2 Grafik zur Laufzeit zuweisen

Im Gegensatz zu den guten alten Windows Forms ist das Laden von Grafiken per Code in WPF etwas mühevoller geworden.

BEISPIEL 13.22: Laden der im vorhergehenden Beispiel erzeugten Ressource

C#
```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            BitmapImage bi = new BitmapImage();
            bi.BeginInit();
            bi.UriSource = new Uri("pack://application:,,,/images/frosch.gif");
```

Alternativ:

```
            bi.UriSource = new Uri("images/frosch.gif",UriKind.Relative);
            bi.EndInit();
            Image5.Source = bi;
        }
```

Der *Source*-Eigenschaft können Sie entweder ein initialisiertes *BitmapImage*-Objekt übergeben, in diesem Fall muss das eigentlich Laden der Datei (egal ob extern oder intern) mit den Methoden *BeginInit* und *EndInit* eingeleitet/beendet werden, oder Sie erzeugen ein *BitmapFrame*-Objekt, wie im folgenden Beispiel gezeigt:

BEISPIEL 13.23: Laden der Ressource per *BitmapFrame*

C#
```
    Image5.Source = BitmapFrame.Create(new Uri("pack://application:,,,/images/frosch.gif"));
```

HINWEIS: In den Microsoft-Dokumentationen findet Sie beim *BitmapImage* noch die Eigenschaften *DecodePixelWidth* und *DecodePixelHeight*. Damit können Sie schon beim Laden des Bildes eine Skalierung durchführen (z.B. entsprechend der Anzeigefläche). Im Speicher wird jetzt nur ein Bild dieser Größe gehalten, eine dauernde Skalierung ist nicht mehr notwendig (weniger Speicher, weniger Rechenzeit).

BEISPIEL 13.24: Maximale Anzeigebreite ist 50, die Originalgröße ist 200

C#
```
...
            BitmapImage bi = new BitmapImage();
            bi.BeginInit();
            bi.DecodePixelWidth = 50;
            bi.UriSource = new Uri("images/frosch.gif",UriKind.Relative);
...
```

HINWEIS: Die Anzeige sollte in diesem Fall mit *Stretch* = *None* erfolgen.

13.8.3 Bild aus Datei laden

Hier müssen wir auf die uralten Dateidialoge zugreifen, WPF hat derzeit noch keine eigenen Dateidialoge.

BEISPIEL 13.25: Laden einer Grafik per Dateidialog

C#

Zunächst den Namespace einbinden:

```
using Microsoft.Win32;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Instanz erzeugen:

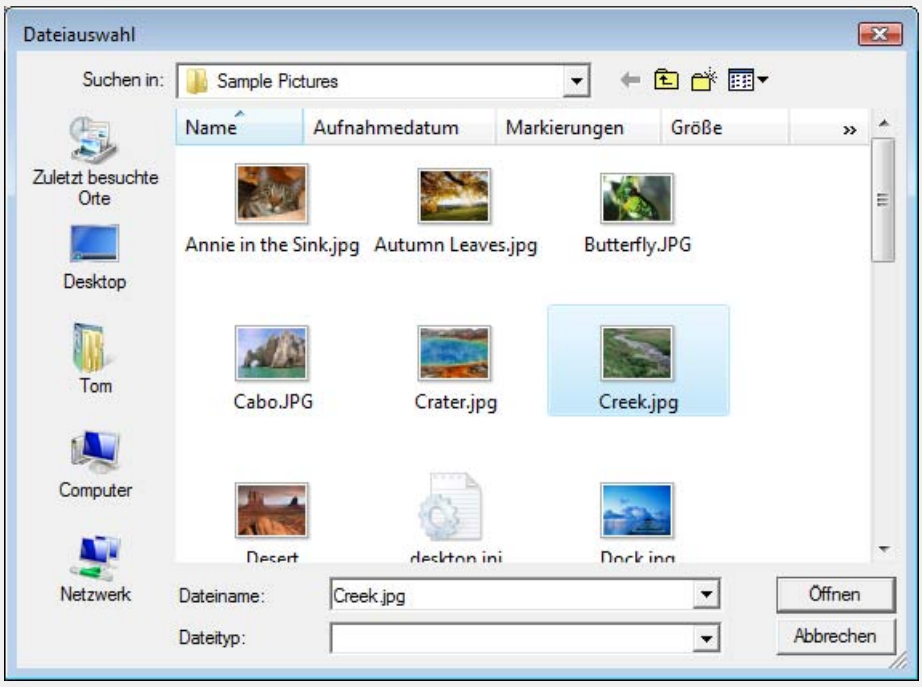
```
            OpenFileDialog Dlg = new OpenFileDialog();
            Dlg.Title = "Dateiauswahl";
            Dlg.DefaultExt = "jpg";
```

Anzeige und Auswertung, wenn Öffnen angeklickt wurde:

```
            if ((bool)Dlg.ShowDialog())
                Image6.Source = BitmapFrame.Create(new Uri(Dlg.FileName));
        }
```

Ergebnis

Der Dateiauswahldialog erscheint beim Öffnen des Fensters:



13.8.4 Die Grafikskalierung beeinflussen

Nicht jede Grafik hat bereits die Größe, die wir für die Anzeigefläche benötigen. Das *Image*-Control stellt aus diesem Grund zwei Eigenschaften zur Verfügung, mit denen Sie das Skalieren konfigurieren können:

- *Stretch* (*None, Fill, Uniform, UniformToFill*)
- *StretchDirection* (*Both, DownOnly, UpOnly*)

BEISPIEL 13.26: Verwendung von *Stretch*

XAML

```
<Grid>
 <Grid.RowDefinitions>
   <RowDefinition Height="*" />
   <RowDefinition Height="*" />
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
   <ColumnDefinition Width="*" />
   <ColumnDefinition Width="*" />
 </Grid.ColumnDefinitions>
```

Originalgröße:

```
<Image Grid.Column="0" Grid.Row="0" Source="Images\Rudi.gif" Stretch="None"/>
```

Skalierung ohne Rücksicht auf Proportionen:

```
<Image Grid.Column="1" Grid.Row="0" Source="Images\rudi.gif" Stretch="Fill"/>
```

Skalierung mit Rücksicht auf Proportionen, längste Seite bestimmt die Größe:

```
<Image Grid.Column="0" Grid.Row="1" Source="Images\rudi.gif" Stretch="Uniform"/>
```

Skalierung mit Rücksicht auf Proportionen, kürzeste Seite bestimmt die Größe:

```
<Image Grid.Column="1" Grid.Row="1" Source="Images\rudi.gif" Stretch="UniformToFill"/>
</Grid>
```

Ergebnis

Wird das *Image* frei positioniert (z.B. *Canvas),* sollten Sie für eine vorgegebene Größe entweder die Höhe **oder** die Breite angeben, nie beides, da es in diesem Fall zu Verzerrungen kommen kann.

HINWEIS: Beachten Sie in diesem Fall auch die Eigenschaften *ActualWidth* und *ActualHeight*, beide liefern erst nach dem Laden des Bildes einen sinnvollen Wert.

13.9 MediaElement

Für die Anzeige von Videos bzw. die Wiedergabe von Audiodateien bietet sich in WPF das *MediaElement* an. Für die Steuerung verwenden Sie die Methoden *Play, Pause* und *Stop*, zuweisen können Sie das Video über die *Source*-Eigenschaft. Doch Achtung:

HINWEIS: Sie können keine als Ressourcen eingebetteten Videos wiedergeben. Kopieren Sie die Videos stattdessen in das Ausgabeverzeichnis (*Build Action=Content, CopyToOutputDirectory=PreserveNewest*).

Nach dem Öffnen der Videodatei stehen Ihnen einige wichtige Eigenschaften zur Verfügung:

- *NaturalDuration (*Laufzeit des Videos)
- *NaturalVideoHeight*, *NaturalVideoWidth* (die Originalabmessungen des Videos)
- *ActualWidth, ActualHeight* (die im Window realisierten Abmessungen des Controls)

Über die Eigenschaft *SpeedRatio* können Sie Einfluss auf die Wiedergabegeschwindigkeit nehmen, Werte größer eins beschleunigen die Wiedergabe, Werte kleiner eins verlangsamen sie.

Wer es gern ruhig mag, der kann *IsMuted* auf *True* setzen, das schont die Ohren.

HINWEIS: Die *LoadedBehavior*-Eigenschaft müssen Sie auf *Manual* setzen, wenn Sie die obigen Methoden zur Steuerung des Controls verwenden wollen.

BEISPIEL 13.27: Verwendung der *MediaElements*

XAML

```
<StackPanel>
  <StackPanel Orientation="Horizontal">
    <Button Click="Button_Click">Start</Button>
    <Button Click="Button_Click_1">Pause</Button>
    <Button Click="Button_Click_2">Stop</Button>
```

Der Lautstärkeregler:

```
    <Slider Name="VSlider" VerticalAlignment="Center" ValueChanged="VSlider_ValueChanged"
                Minimum="0" Maximum="1" Value="0.5" Width="70"/>
```

Die Geschwindigkeit:

```
    <Slider Name="SSlider" VerticalAlignment="Center" ValueChanged="SSlider_ValueChanged"
                 Value="1" Maximum="5" Minimum="0.1" Width="70" />
  </StackPanel>
```

Das eigentliche *MediaElement*:

```
  <MediaElement LoadedBehavior="Manual" UnloadedBehavior="Stop" Width="400" Name="Media1"
   Stretch="Uniform" Source="butterfly.wmv" MediaOpened="Media1_MediaOpened"
   MediaEnded="Media1_MediaEnded">
  </MediaElement>
```

BEISPIEL 13.27: Verwendung der *MediaElements*

C#

Der Ereigniscode:

```
private void Button_Click(object sender, RoutedEventArgs e)
{  Media1.Play(); }

private void Button_Click_1(object sender, RoutedEventArgs e)
{ Media1.Pause(); }

private void Button_Click_2(object sender, RoutedEventArgs e)
{ Media1.Stop();  }
```

Lautstärke regeln:

```
private void VSlider_ValueChanged(object sender,
          RoutedPropertyChangedEventArgs<double> e)
{ if (Media1 != null) Media1.Volume = (double)VSlider.Value; }
```

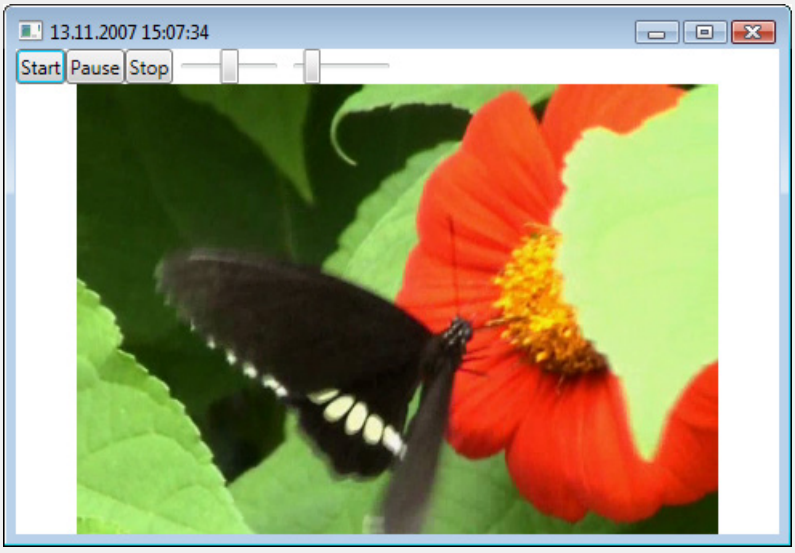
Geschwindigkeit regeln:

```
private void SSlider_ValueChanged(object sender,
           RoutedPropertyChangedEventArgs<double> e)
{ if (Media1!=null) Media1.SpeedRatio = (double)SSlider.Value; }
```

Nach dem Öffnen des Videos bzw. am Ende:

```
private void Media1_MediaOpened(object sender, RoutedEventArgs e)
{ }

private void Media1_MediaEnded(object sender, RoutedEventArgs e)
{  }
```

Ergebnis



13.10 Slider, ScrollBar

Im Folgenden wollen wir uns mit zwei Controls beschäftigen, die für die Eingabe von Werten innerhalb eines bestimmten Wertebereichs geeignet sind.

13.10.1 Slider

Im vorhergehenden Abschnitt hatten wir bereits auf das *Slider*-Control zurückgegriffen, um die Lautstärke und die Abspielgeschwindigkeit zu regeln (siehe obige Abbildung). Damit dürfte auch schon der Verwendungszweck erkennbar sein:

- Auswahl eines Wertes durch Verschieben des Reglers (*Value*)
- Vorgabe eines minimalen Wertes (*Minimum)*
- Vorgabe eines maximalen Wertes (*Maximum)*

Beachten Sie, dass es sich beim *Value* um einen *Double*-Wert handelt, Sie also gegebenenfalls eine Typisierung/Rundung durchführen müssen, um den Wert in Ihrem Programm sinnvoll nutzen zu können. Änderungen der Auswahl können Sie über das *ValueChanged*-Ereignis auswerten.

HINWEIS: Natürlich können Sie den *Slider* auch als Ausgabemedium zur Wertanzeige missbrauchen.

Neben den drei genannten Eigenschaften bieten sich auch noch weitere Möglichkeiten zur Konfiguration an:

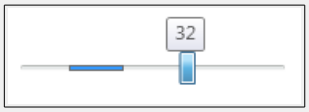
- **Ausrichtung**
  Verwenden Sie *Orientation*, um die Ausrichtung des Controls zu beeinflussen (*Vertical, Horizontal*).
- **Anzeige von Werten**
  Hier bietet sich die Verwendung von Tooltips an. Setzen Sie *AutoToolTipPlacement* auf *BottomRight* oder *TopLeft,* um den jeweils aktuellen Wert beim Verschieben des Sliders anzuzeigen. Die Genauigkeit bestimmen Sie mit *AutoToolTipPrecision*.
- **Werteverlauf**
  Standardmäßig finden Sie die kleinsten Werte links bzw. unten. Sollen sich diese rechts bzw. oben befinden, müssen Sie *IsDirectionReversed* auf *True* setzen.
- **Markierungsbereich**
  Mit *IsSelectionRangeEnabled* aktivieren Sie die Anzeige eines Markierungsbereichs. Den Bereich selbst definieren Sie über *SelectionStart* und *SelectionEnd.*
- **Markierung, Skala**
  Mit *TickPlacement* (*Both, BottomRight, TopLeft*) können Sie die Anzeige von Markierungsstrichen einschalten. Da standardmäßig die Striche für jeden ganzzahligen Wert gesetzt werden, sollten Sie mit *TickFrequency* ein anderes Intervall festlegen.

- **Schrittweite**
  Haben Sie *TickFrequency* festgelegt, können Sie mit *IsSnapToTickEnabled=True* erreichen, dass nur Werte genau an den Markierungsstrichen ausgewählt werden können.

BEISPIEL 13.28: Horizontaler *Slider* mit Markierungsbereich und Tooltip

XAML
```
<Slider  AutoToolTipPlacement="TopLeft"  IsSelectionRangeEnabled="True" SelectionStart="10"
         SelectionEnd="20" Minimum="1" Maximum="50" Value="25" Canvas.Left="11"
         Canvas.Top="55" Height="26" Name="slider1" Width="179" />
```

Ergebnis



BEISPIEL 13.29: Vertikaler *Slider* mit Markierungsstrichen und zwangsweiser Auswahl (0, 10, 20 .. 100)

XAML
```
<Slider  TickPlacement="Both" TickFrequency="10" IsSnapToTickEnabled="True"
         AutoToolTipPlacement="BottomRight" Width="43"  Orientation="Vertical"
         Canvas.Left="220" Canvas.Top="21" Height="111" Name="slider2" Maximum="100" />
```

Ergebnis

13.10.2 ScrollBar

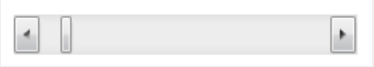
Ähnlich wie beim *Slider* bietet ein *ScrollBar* die Möglichkeit, Werte (*Value*) aus einem Wertebereich (*Minimum, Maximum)* auszuwählen. Scrollbars können mit *Orientation* ebenfalls horizontal oder vertikal ausgerichtet werden. Die Auswertung kann mit dem Ereignis *Scroll* erfolgen, alternativ kommt jedoch meist eine direkte Datenbindung (siehe ab Seite 787) in Frage.

Im Gegensatz zum *Slider* fehlen jedoch alle weiteren Möglichkeiten zur Konfiguration (Wertanzeige, Skala etc.).

BEISPIEL 13.30: Verwendung *ScrollBar*

XAML
```
<ScrollBar Maximum="10" Scroll="ScrollBar_Scroll" Orientation="Horizontal" Canvas.Left="42"
Canvas.Top="182" Height="25" Name="scrollBar1" Width="214" />
```

Ergebnis



13.11 ScrollViewer

Nicht genug der Scroller, mit dem *ScrollViewer* können Sie direkt ein im *Content* enthaltenes Container-Control horizontal und vertikal verschieben, wenn es nicht in den sichtbaren Clientbereich des ScrollViewers passt. Ob und wann die ScrollBars angezeigt werden, entscheiden Sie mit den Eigenschaften *HorizontalScrollBarVisibility* und *VerticalScrollBarVisibility*.

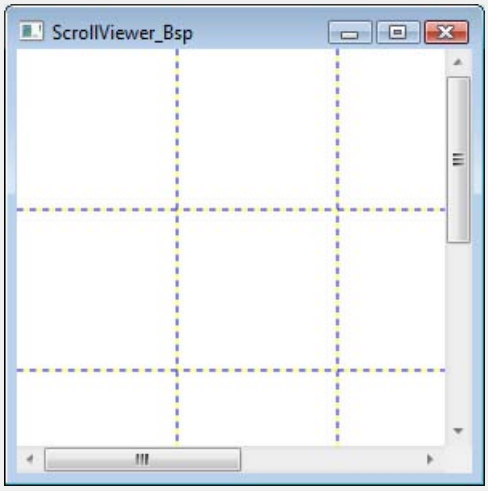
Häufiger Verwendungszweck dieses Controls ist die Darstellung von Grafiken, die zu groß für die Anzeige im Fenster sind.

BEISPIEL 13.31: Ein *Grid* mit fester Größe soll in einem *ScrollViewer* angezeigt werden

XAML

```
<ScrollViewer Name="scrollViewer1" HorizontalScrollBarVisibility="Visible"
              VerticalScrollBarVisibility="Visible">
  <Grid Width="500" Height="500" ShowGridLines="True">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
      <RowDefinition Height="*" />
...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="*" />
...
    </Grid.ColumnDefinitions>
  </Grid>
</ScrollViewer>
```

Ergebnis



Entsprechend der Größe des enthaltenen Controls (*Grid*) wird der Scrollbereich angepasst.

13.12 Menu, ContextMenu

Nachdem wir uns schon eine Reihe der geläufigsten Controls angesehen haben, wollen wir uns endlich an ein wesentliches Oberflächenelement wagen, das wohl in fast keinem Programm fehlt: die Menüleiste. An dieser Stelle gleich ein wichtiger Hinweis, damit Ihre Ambitionen nicht zu weit gehen:

HINWEIS: WPF unterstützt keine MDI-Anwendungen!

Eine Alternative finden Sie jedoch hier:

LINK: http://wpfmdi.codeplex.com/

13.12.1 Menu

Ein eigenes Menü erstellen Sie recht einfach per XAML mit dem <*Menu*>-Element, in das Sie ineinander geschachtelte <*MenuItem*>-Elemente einfügen. Zusätzlich können Sie zum Gruppieren noch <*Separator*>-Elemente verwenden. Die Auswertung können Sie, wie nicht anders zu erwarten, mit dem *Click*-Ereignis vornehmen.

BEISPIEL 13.32: Ein einfaches Menü erstellen

XAML

```
<Window x:Class="BSP_Controls.Menu_ToolBar_StatusBar"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Menu_ToolBar_StatusBar" Height="300" Width="300">
```

Zunächst ein *DockPanel,* um das Menü auch oben zu verankern:

```
 <DockPanel>
```

Hier das Menü:

```
   <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
```

Die erste Gruppe von *MenuItem*s:

```
     <MenuItem Header="_Datei">
       <MenuItem Header="Neu"/>
       <MenuItem Header="Öffnen"/>
       <MenuItem Header="Sichern"/>
```

Hier ein Separator (Trennstrich):

```
       <Separator/>
       <MenuItem Header="Ende"/>
     </MenuItem>
```

Noch zwei Hauptmenüpunkte:

```
     <MenuItem Header="_Bearbeiten"/>
     <MenuItem Header="_Hilfe"/>
```

BEISPIEL 13.32: Ein einfaches Menü erstellen

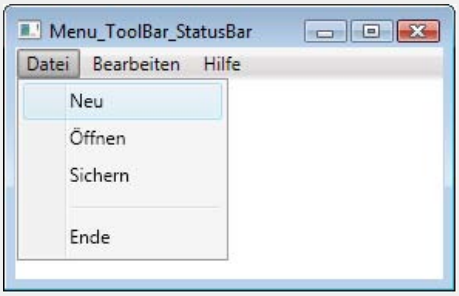
```
    </Menu>
```

Das folgende *StackPanel* füllt den verbliebenen Platz im *DockPanel*:

```
    <StackPanel>
    </StackPanel>
  </DockPanel>
</Window>
```

Ergebnis

Im Endergebnis dürfte folgendes *Window* angezeigt werden:



13.12.2 Tastenkürzel

Wie Sie sicher bereits im obigen Beispiel bemerkt haben, werden die Alt-Tasten-Shortcuts durch einen Unterstrich markiert. Möchten Sie erweiterte Tastenkombinationen verwenden, können Sie diese über das *InputGestureText*-Attribut einem Menüpunkt zuweisen.

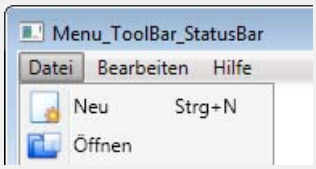
BEISPIEL 13.33: Verwendung von *InputGestureText*

XAML

```
<MenuItem Header="Neu" InputGestureText="Strg+N" Click="MenuItem_Click">
  <MenuItem.Icon>
    <Image Source="Images/filenew.png" Width="22"/>
  </MenuItem.Icon>
</MenuItem>
```

Ergebnis

Doch ein erster Test wird Sie auf den harten Boden der Realität zurückholen, die Tastenfolge wird zwar im Menü angezeigt



... eine Auswertung des Shortcuts erfolgt jedoch nicht. Hier müssen Sie, wie im Programmier-mittelalter, die Tastenfolgen im Window auswerten.

HINWEIS: Alternativ können Sie in WPF sogenannte Commands verwenden, wir gehen in Kapitel 14 (ab Seite 748) darauf ein.

13.12.3 Grafiken

Bisher sehen unsere Menüpunkte noch recht trist aus, es fehlen die bekannten Grafiken. Doch auch dies ist kein Problem, WPF bietet zwei Varianten für die Anzeige von Grafiken an:

- Ein zusätzliches Icon links neben dem Texteintrag
- Eine beliebige Kombination von Grafik/Elementen innerhalb des Menüeintrags

BEISPIEL 13.34: Einblenden eines zusätzlichen Icons links neben dem Menüeintrag

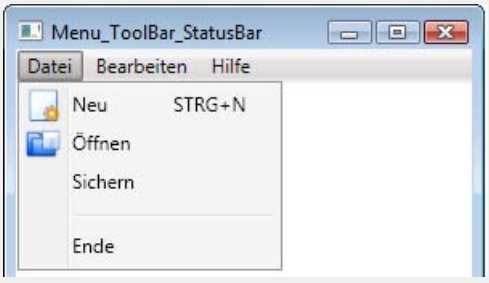
XAML

```
  <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
    <MenuItem Header="_Datei">
      <MenuItem Header="Neu">
```

Hier weisen Sie die Grafik zu:

```
        <MenuItem.Icon>
          <Image Source="Images/filenew.png" Width="22"/>
        </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="Öffnen">
        <MenuItem.Icon>
          <Image Source="Images/fileopen.png" Width="22"/>
        </MenuItem.Icon>
      </MenuItem>
...
```

Ergebnis



BEISPIEL 13.35: Ein grafischer Menüeintrag mit freier Gestaltung (zusätzliche *ComboBox*)

XAML

```
  <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
    <MenuItem Header="_Datei">
...
      <MenuItem>
```

BEISPIEL 13.35: Ein grafischer Menüeintrag mit freier Gestaltung (zusätzliche *ComboBox*)

Mit einem *MenuItem.Header*-Element bietet sich die Möglichkeit, beliebige Elemente in einem Menüeintrag unterzubringen:

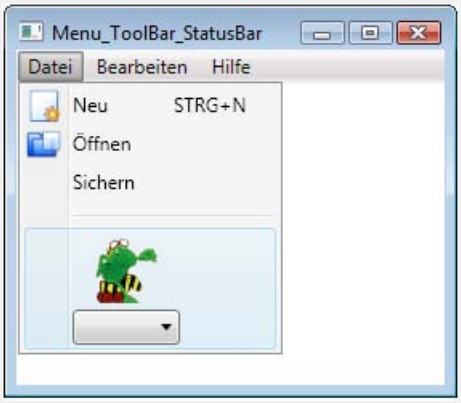
```
<MenuItem.Header>
```

Fügen Sie zunächst ein Layout-Element ein, danach ist die Positionierung von Grafiken oder anderen Elementen ein Kinderspiel:

```
          <StackPanel>
            <Image Source="images/frosch.gif" Height="40" Margin="4"/>
            <ComboBox>
              <ComboBoxItem>Zeile 1</ComboBoxItem>
              <ComboBoxItem>Zeile 2</ComboBoxItem>
              <ComboBoxItem>Zeile 3</ComboBoxItem>
              <ComboBoxItem>Zeile 4</ComboBoxItem>
              <ComboBoxItem>Zeile 5</ComboBoxItem>
            </ComboBox>
          </StackPanel>
        </MenuItem.Header>
      </MenuItem>
...
```

Ergebnis

Die folgende Abbildung zeigt des Ergebnis unserer Bemühungen, ob Sie eine *ComboBox* unbedingt im Menü unterbringen müssen, ist allerdings fraglich.



13.12.4 Weitere Möglichkeiten

Neben den bereits vorgestellten Möglichkeiten bietet sich mit den Eigenschaften *IsCheckable* und *IsChecked* die Möglichkeit, Optionen innerhalb des Menüs zu aktivieren bzw. zu deaktivieren.

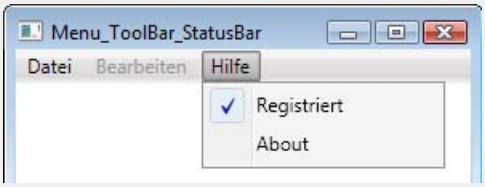
Den Menüpunkt selbst können Sie mit *IsEnabled=False* deaktivieren.

BEISPIEL 13.36: *IsCheckable* und *IsEnabled*

XAML

```
    <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
      <MenuItem Header="_Bearbeiten" IsEnabled="False"/>
      <MenuItem Header="_Hilfe">
        <MenuItem Header="Registriert" IsCheckable="True" IsChecked="True"/>
        <MenuItem Header="About" IsEnabled="True"/>
      </MenuItem>
    </Menu>
```

Ergebnis



13.12.5 ContextMenu

Auf das Kontextmenü trifft weitgehend das bereits Gesagte zu, mit dem Unterschied, dass ein *ContextMenu* einem spezifischen Control zugeordnet wird.

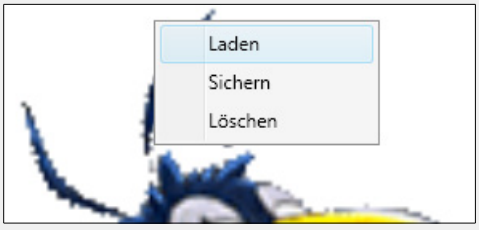
BEISPIEL 13.37: Erzeugen eines Kontextmenüs

XAML

```
    <Image Source="Images/rudi.gif" >
      <Image.ContextMenu>
        <ContextMenu>
          <MenuItem Header="Laden" />
          <MenuItem Header="Sichern" />
          <MenuItem Header="Löschen" />
        </ContextMenu>
      </Image.ContextMenu>
    </Image>
```

Ergebnis

Das erzeugte Kontextmenü:



13.13 ToolBar

WPF-Anwendungen können eine Werkzeugleiste mit dem *ToolBar*-Control implementieren, allerdings haben Sie es hier nicht, wie vielleicht erwartet, mit einem hoch komplexen Control, sondern "lediglich" einem Container für die bereits aufgeführten Controls zu tun. Das beginnt bereits beim Einfügen des Controls, Sie als Programmierer müssen sich darum kümmern, dass der *ToolBar* auch dahin kommt wo er hin soll. Verwenden Sie dazu ein *DockPanel* und richten Sie das Control an der gewünschten Seite aus.

HINWEIS: Optional können Sie einen/mehrere Toolbar(s) in einem *ToolBarTray* unterbringen, hier ist dann ein Positionieren und Verschieben möglich.

BEISPIEL 13.38: Ein einfacher *ToolBar* mit drei Schaltflächen und einer *ComboBox*

XAML

```
<Window x:Class="BSP_Controls.Menu_ToolBar_StatusBar"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Menu_ToolBar_StatusBar" Height="300" Width="300">

 <DockPanel>
   <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
   </Menu>
```

Zunächst der äußere Rahmen (ein *ToolBarTray*):

```
   <ToolBarTray DockPanel.Dock="Top">
```

Dann der eigentliche *Toolbar*:

```
     <ToolBar>
```

Und jetzt passiert das Gleiche wie in den anderen Layout-Containern (z.B. *StackPanel* mit horizontaler Ausrichtung):

```
      <Button Width="30" Height="30">
        <Image Source="Images/filenew.png"/>
      </Button>
      <Button Width="30" Height="30">
        <Image Source="Images/fileopen.png"/>
      </Button>
      <Button Width="30" Height="30">
        <Image Source="Images/filesave.png"/>
      </Button>
```

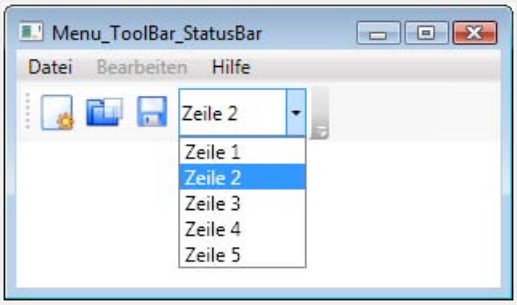
Das funktioniert natürlich auch mit einer *ComboBox*:

```
      <ComboBox Width="80" Height="30">
        <ComboBoxItem>Zeile 1</ComboBoxItem>
        <ComboBoxItem>Zeile 2</ComboBoxItem>
        <ComboBoxItem>Zeile 3</ComboBoxItem>
```

BEISPIEL 13.38: Ein einfacher *ToolBar* mit drei Schaltflächen und einer *ComboBox*

```
        <ComboBoxItem>Zeile 4</ComboBoxItem>
        <ComboBoxItem>Zeile 5</ComboBoxItem>
      </ComboBox>
    </ToolBar>
  </ToolBarTray>
</Window>
```

Ergebnis



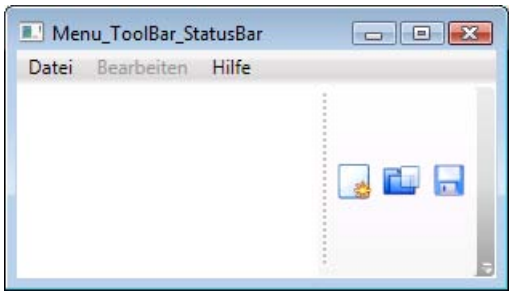
Der ToolBarTray

Warum fügen wir eigentlich den *ToolBar* in einen *ToolBarTray* ein? Die Antwort finden Sie schnell, wenn Sie beispielsweise den *ToolBar* am rechten oder linken Rand des Windows platzieren möchten.

Mit den folgenden Anweisungen

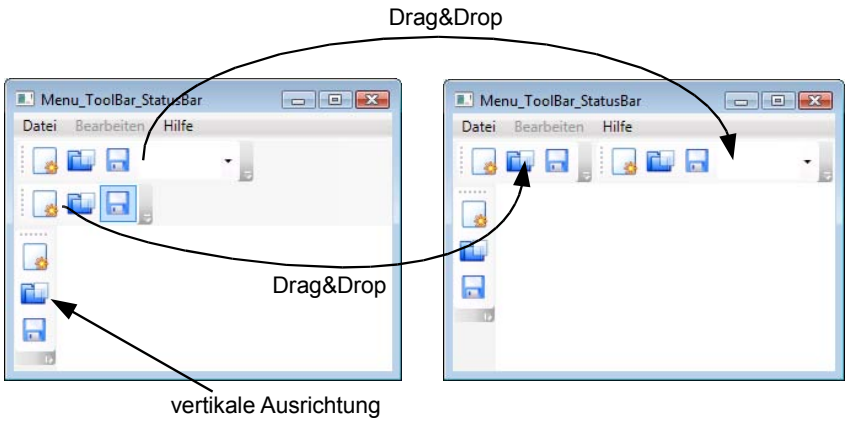
```
  <ToolBar DockPanel.Dock="Right">
    <Button Width="30" Height="30">
      <Image Source="Images/filenew.png"/>
    </Button>
...
```

... wird zwar der *ToolBar* rechts platziert, es erfolgt aber keine vertikale Ausrichtung. Eine entsprechende Eigenschaft ist nicht vorhanden.



Auch das Platzieren (per Code oder Drag&Drop) mehrerer einzelner *ToolBar*s wird so schnell zum Geduldspiel.

Fügen Sie hingegen die einzelnen *ToolBar*-Controls in einen *ToolBarTray* ein, können Sie diesen per *Orientation*-Eigenschaft problemlos vertikal ausrichten. Auch das Platzieren der einzelnen *ToolBar*s per Drag&Drop durch den Anwender ist jetzt möglich:

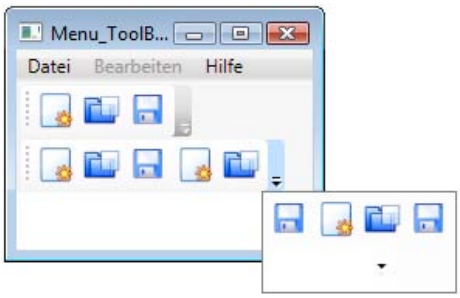


Mit der Verwendung des *ToolBarTray* ergeben sich jedoch noch weitere Möglichkeiten:

- Mit der *ToolBar*-Eigenschaft *Band* legen Sie die Zeile innerhalb des *ToolbarTray* fest (siehe obige Abbildung mit zweizeiligem *ToolBarTray*.
- Die Reihenfolge mehrerer *ToolBar*-Controls innerhalb einer Zeile legen Sie über die Eigenschaft *BandIndex* fest.
- Möchten Sie das Verschieben von *ToolBar*s innerhalb des *ToolBarTray* verhindern, setzen Sie deren *IsLocked* auf *True*.

HINWEIS: Blenden Sie nicht benötigte *ToolBar*-Controls mit *Visibility=Collapsed* aus, wenn Sie den vom *ToolBar* benötigten Platz freigeben möchten. Andernfalls genügt ein *Hidden*, der *ToolBarTray* bleibt in diesem Fall sichtbar.

Was passiert eigentlich, wenn nicht genügend Platz für die komplette Darstellung des *ToolBar*s bleibt? In diesem Fall werden normalerweise die nicht mehr darstellbaren Controls in einem extra Menü (Überlaufbereich) angezeigt:



Sie als Programmierer können darüber entscheiden, wann welche Controls im Überlaufbereich angezeigt werden sollen. Dazu steht die angehängte Eigenschaft *ToolBar.OverflowMode* zur Verfügung. Für jedes einzelnen Control können Sie festlegen, ob es niemals (*Never*) bei Bedarf (*AsNeeded*) oder immer (*Always*) im Überlaufbereich angezeigt wird.

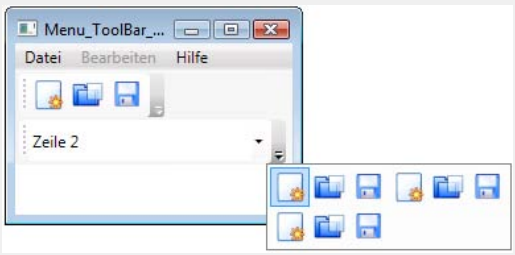
BEISPIEL 13.39: Eine *ComboBox* soll nie im Überlaufbereich angezeigt werden

XAML

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar Band="1" >
    <Button Width="30" Height="30">
      <Image Source="Images/filenew.png"/>
    </Button>
    <ComboBox ToolBar.OverflowMode="Never" Width="80" Height="30">
      <ComboBoxItem>Zeile 1</ComboBoxItem>
      <ComboBoxItem>Zeile 2</ComboBoxItem>
      <ComboBoxItem>Zeile 3</ComboBoxItem>
      <ComboBoxItem>Zeile 4</ComboBoxItem>
      <ComboBoxItem>Zeile 5</ComboBoxItem>
    </ComboBox>
  </ToolBar>
```

Ergebnis

Obwohl die *ComboBox* das letzte Control innerhalb des *ToolBar*s ist, wird diese jetzt niemals im Überlaufbereich erscheinen. Stattdessen wird sie im "Notfall" einfach abgeschnitten:



Bleibt eine letzte Frage: wie reagiere ich auf Ereignisse? Die Antwort ist ganz einfach: wie bei jedem anderen Control auch, d h. mit *Click* oder *SelectionChanged*.

13.14 StatusBar, ProgressBar

Nach dem Menü und der Werkzeugleiste fehlt zu einer "kompletten" Oberfläche meist noch eine Statusleiste für die Ausgabe von Programminformationen.

13.14.1 StatusBar

Wie auch beim Menü oder bei der Werkzeugleiste sind **Sie** dafür verantwortlich, den *StatusBar* an den gewünschten Platz zu bringen. Üblicherweise werden Sie diese Aufgabe mit einem *DockPanel* realisieren, dieses benötigen Sie sowieso für die beiden anderen genannten Controls.

Zwei Varianten zur Nutzung des *StatusBar* bieten sich an:

- Sie verwenden innerhalb des *StatusBar*s so genannte *StatusBarItems,* in die Sie wiederum per Layout-Control (*StackPanel, Grid)* weitere Controls (*Label*, *TextBox, Button* etc.) einfügen können, oder
- Sie verzichten auf die *StatusBarItem*s und setzen einfach die benötigen Controls in den Content des *StatusBar*.

Variante eins hat den Vorteil der Übersichtlichkeit, der einfachen Gruppierbarkeit von Controls sowie der Möglichkeit, die Controls auch recht einfach am rechten Rand auszurichten. Dazu verwenden Sie die *DockPanel.Dock*-Eigenschaft des *StatusBarItems*[1].

HINWEIS: Meist müssen Sie die Höhe/Breite von enthaltenen Controls vorgeben, andernfalls sind diese nicht sichtbar.

BEISPIEL 13.40: Verwendung des *StatusBar* mit rechter Ausrichtung eines *ProgressBar*

XAML

```
<Window x:Class="BSP_Controls.Menu_ToolBar_StatusBar" Title="Menu_ToolBar_StatusBar"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Height="300" Width="300">
 <DockPanel>
...
```

Zunächst Ausrichtung im Fenster:

```
  <StatusBar DockPanel.Dock="Bottom">
```

Ein paar einfach positionierte Controls:

```
    <Label Content="Suchtext"/>
    <TextBox Width="50">*</TextBox>
    <Button>Suchen</Button>
```

Sie könne auch einen Trenner einfügen:

```
    <Separator/>
```

Wir richten ein *StatusBarItem* am rechten Rand aus

```
    <StatusBarItem DockPanel.Dock="Right">
```

... und fügen einen *ProgressBar* ein:

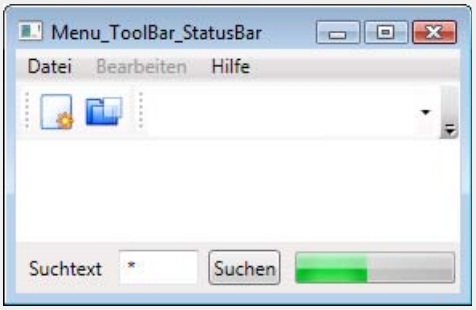
```
      <ProgressBar Width="100" Height="20" Value="45"/>
    </StatusBarItem>
  </StatusBar>
  <StackPanel>
  </StackPanel>
 </DockPanel>
</Window>
```

[1] Der *StatusBar* ist ein verkapptes *DockPanel*.

BEISPIEL 13.40: Verwendung des *StatusBar* mit rechter Ausrichtung eines *ProgressBar*

Ergebnis

Die Laufzeitansicht unseres *StatusBar*s:



13.14.2 ProgressBar

Nicht alles läuft so schnell wie wir es gern hätten, und so hat auch in Zeiten von Quad-Core-Prozessorsystemen der gute alte Fortschrittsbalken seine Daseinsberechtigung behalten.

In WPF-Anwendungen verwenden Sie dazu das *ProgressBar*-Control, das zwei Modi kennt:

- *IsIndeterminate=True*,
  der Fortschrittsbalken stellt eine endlose Animation dar, um "Bewegung" zu zeigen[1]:

- *IsIndeterminate=False*,
  der bekannte Fortschrittsbalken, der mit *Minimum, Maximum* und *Value* konfiguriert wird:

Das Control selbst kann horizontal oder vertikal angeordnet werden (*Orientation*), bester Aufbewahrungsort für den *ProgressBar* dürfte sicher der *StatusBar* sein.

BEISPIEL 13.41: Verwendung des *ProgressBar*-Controls

XAML

```
<StatusBar DockPanel.Dock="Bottom">
  <Label Content="Suchtext"/>
  <TextBox Width="50">*</TextBox>
  <StatusBarItem DockPanel.Dock="Right">
    <ProgressBar Width="100" Height="20" Value="45"/>
  </StatusBarItem>
  <Button>Suchen</Button>
  <Separator/>
</StatusBar>
```

[1] Das ideale Control für den Datei kopieren-Dialog unter Windows (gleich fertig, gleich fertig ...)

13.15 Border, GroupBox, BulletDecorator

Nachdem wir uns bisher mit ganz praktischen Controls beschäftigt haben, wollen wir auch etwas für die Ordnung bzw. fürs Auge vorstellen. Die im Folgenden gezeigten Controls haben eigentlich nur den Zweck, die Optik des Programms zu verbessern.

13.15.1 Border

Möchten Sie einen (fast) frei definierbaren Rahmen um einzelne Controls zeichnen, bietet sich die Verwendung eines *Border*-Controls an.

Fügen Sie in den *Content* dieses Controls ein eigenes Layout-Control ein, um die darin enthaltenen Controls sinnvoll anzuordnen. Bei der Konfiguration des Borders können Sie neben Vorder- und Hintergrundfarbe/-muster auch die Rahmenbreite (*BorderThickness*) und den Eckradius (*CornerRadius*) festlegen. Letzteres kann für jede Ecke einzeln erfolgen, geben Sie in diesem Fall einfach vier statt einem Wert an (links oben, rechts oben, rechts unten, links unten). Gleiches gilt auch für die Rahmenbreite (links, oben, rechts, unten).

Die Abstände zum Rand bzw. zum Inhalt legen Sie wie gewohnt mit *Margin* bzw. *Padding* fest.

BEISPIEL 13.42: Verwendung *Border*

XAML

```
<UniformGrid Margin="4" Columns="3" Rows="3">
```

Variante 1:

```
  <Border Margin="4" BorderThickness="2" CornerRadius="12" BorderBrush="Black"/>
```

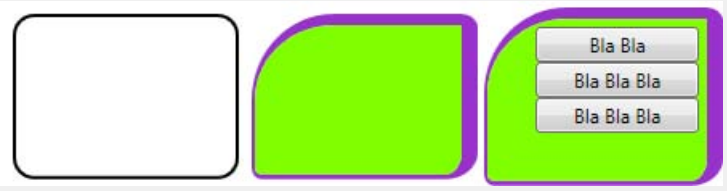
Variante 2:

```
  <Border Margin="4" BorderThickness ="2,7,10,3" CornerRadius="50,5,15,5"
          Background="Chartreuse" BorderBrush="DarkOrchid"/>
```

Variante 3:

```
  <Border Padding="30,5,5,5" BorderThickness="2,7,10,3" CornerRadius="50,5,15,5"
          Background="Chartreuse" BorderBrush="DarkOrchid">
    <StackPanel>
      <Button>Bla Bla</Button>
      <Button>Bla Bla Bla</Button>
      <Button>Bla Bla Bla</Button>
    </StackPanel>
  </Border>
</UniformGrid>
```

Ergebnis



13.15.2 GroupBox

Ein naher Verwandter des *Border*-Controls ist die *GroupBox*. Statt der Möglichkeit, den Eckradius zu beeinflussen, können Sie hier eine Kopfzeilenbeschriftung (*Header)* realisieren. Dabei müssen Sie sich nicht auf reinen Text beschränken, Sie können auch andere Controls und damit zum Beispiel auch Grafiken etc. in den Kopfbereich einfügen.

BEISPIEL 13.43: Verwendung der *GroupBox*

XAML

```
<UniformGrid Margin="4" Columns="3" Rows="3">
```

Zunächst die einfachste Variante mit reinem Text:

```
<GroupBox Header="Stammdaten" BorderThickness="2" BorderBrush="HotPink" Padding="5">
  <StackPanel>
    <Button>Bla Bla</Button>
    <Button>Bla Bla Bla</Button>
    <Button>Bla Bla Bla</Button>
  </StackPanel>
</GroupBox>
```

Hier die Variante mit einer *CheckBox* im Header:

```
<GroupBox BorderThickness="2" BorderBrush="HotPink" Padding="5" Background="AliceBlue">
  <GroupBox.Header>
    <CheckBox>Fahrzeugdaten</CheckBox>
  </GroupBox.Header>
    <StackPanel>
    <Label>Kennzeichen</Label>
    <TextBox>LOS LW 77</TextBox>
  </StackPanel>
</GroupBox>
```

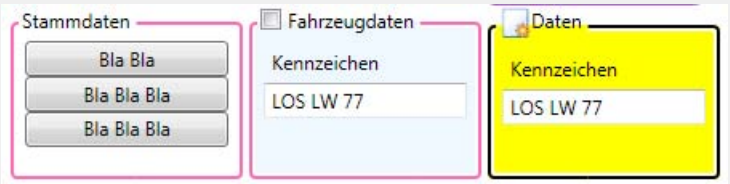
Es geht auch mit einer Grafik und Text:

```
<GroupBox BorderThickness="2" BorderBrush="Black" Padding="5" Background="Yellow">
  <GroupBox.Header>
    <StackPanel Orientation="Horizontal">
        <Image Source="Images/Filenew.png"  Width="20" />
        <TextBlock> Daten</TextBlock>
    </StackPanel>
  </GroupBox.Header>
  <StackPanel>
    <Label>Kennzeichen</Label>
    <TextBox>LOS LW 77</TextBox>
  </StackPanel>
</GroupBox>
<StackPanel>
</UniformGrid>
```

BEISPIEL 13.43: Verwendung der *GroupBox*

Ergebnis

Die drei Varianten:



HINWEIS: Setzen Sie *RadioButton*s in eine *GroupBox,* um diese als logische Gruppe zu organisieren. So brauchen Sie nicht die *GroupName*-Eigenschaft festzulegen, und die Zusammenhänge fallen auch optisch auf.

13.15.3 BulletDecorator

Dieses Control dient der Anzeige von Aufzählungen, d h., einem Aufzählungszeichen/-grafik (*BulletDecorator.Bullet*) und einem beschreibenden Text[1].

BEISPIEL 13.44: Verwendung *BulletDecorator*

XAML

```
<UniformGrid Margin="4" Columns="3" Rows="3">
```

Variante 1 (Ellipse + Text):

```
  <StackPanel>
    <BulletDecorator Margin="5">
```

Das Aufzählungszeichen definieren:

```
      <BulletDecorator.Bullet>
        <Ellipse Width="10" Height="10" Fill="Red" />
      </BulletDecorator.Bullet>
```

Den Textteil definieren:

```
      <TextBlock Margin="5 0 0 0" TextWrapping="NoWrap">
        Zeile 1
      </TextBlock>
    </BulletDecorator>
```

Und jetzt die nächste Aufzählung:

```
    <BulletDecorator Margin="5">
      <BulletDecorator.Bullet>
        <Ellipse Width="10" Height="10" Fill="Red" />
      </BulletDecorator.Bullet>
```

[1] Mit einem *Grid* können Sie derartige Aufgaben sicher einfacher realisieren.

BEISPIEL 13.44: Verwendung *BulletDecorator*

```
      <TextBlock Margin="5 0 0 0" TextWrapping="NoWrap">
        Zeile 2
      </TextBlock>
    </BulletDecorator>
  </StackPanel>
```

Variante 2 (mit *CheckBox):*

```
  <BulletDecorator Margin="0,5,0,0">
    <BulletDecorator.Bullet>
      <CheckBox/>
    </BulletDecorator.Bullet>
    <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left" Margin="5,0,0,0">
      Mit CheckBox
    </TextBlock>
  </BulletDecorator>
```

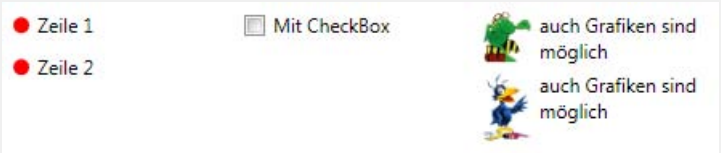
Variante 3 (mit Grafik):

```
  <StackPanel>
    <BulletDecorator Margin="0,5,0,0">
      <BulletDecorator.Bullet>
        <Image Source="Images/frosch.gif" Width="30"/>
      </BulletDecorator.Bullet>
      <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left"
                 Margin="5,0,0,0">
        auch Grafiken sind möglich
      </TextBlock>
    </BulletDecorator>
```

Und hier der nächste Eintrag:

```
    <BulletDecorator Margin="0,5,0,0">
      <BulletDecorator.Bullet>
        <Image Source="Images/rudi.gif" Width="30"/>
      </BulletDecorator.Bullet>
      <TextBlock Width="100" TextWrapping="Wrap" HorizontalAlignment="Left"
                 Margin="5,0,0,0">
        auch Grafiken sind möglich
      </TextBlock>
    </BulletDecorator>
  </StackPanel>
</UniformGrid>
```

Ergebnis



13.16 RichTextBox

Um es gleich vorweg zu sagen, dieses Control ist nicht mit seinen doch recht rudimentär entwickelten Vorgängern bei den Windows Forms oder den Win32-Controls zu vergleichen. Aufgabe des Controls ist das komfortable Bearbeiten von formatierten Dokumenten. Basis ist dabei ein so genanntes *FlowDocument,* in dem Sie neben den bekannten Zeichenformatierungen (Fett, Kursiv, Unterstrichen, Hochstellen ...) auch wesentlich komplexere Formate, wie Grafiken, Absätze, Listen, Tabellen etc., realisieren können.

Das Control unterstützt beim Im-/Export neben den bekannten Formaten RTF, TEXT auch XAML und XAML-Packages.

HINWEIS: Statt Sie auf den folgenden Seiten mit endlosen Eigenschaftsauflistungen zu foltern (und davon hat das Control reichlich), wollen wir es uns "aufgaben-orientiert" ansehen.

13.16.1 Verwendung und Anzeige von vordefiniertem Text

Eine *RichTextBox* ist mit dem gleichnamigen Element schnell definiert, doch was ist mit dem Inhalt?

Einzig gültiges Element im Content der *RichTextBox* ist ein *FlowDocument,* auf dieses können Sie zur Laufzeit über die *Document*-Eigenschaft des Controls zugreifen. Innerhalb dieses *FlowDocuments* können Sie

- *BlockUIContainer*-
- *List*-
- *Paragraph*-
- *Section*-
- oder *Table*-Elemente

definieren. Mehr dazu ab Seite 687 (*FlowDocument),* an dieser Stelle beschränken wir uns auf einen simplen Absatz mit einfachen Formatierungen.

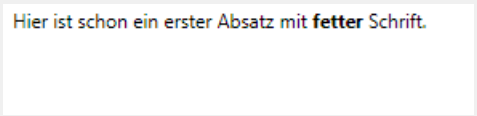
BEISPIEL 13.45: Eine *RichTextBox* mit vordefiniertem Text anzeigen

XAML

```
<RichTextBox>
  <FlowDocument>
    <Paragraph>
        Hier ist schon ein erster Absatz mit <Bold>fetter</Bold> Schrift.
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

BEISPIEL 13.45: Eine *RichTextBox* mit vordefiniertem Text anzeigen

Ergebnis



BEISPIEL 13.46: (Fortsetzung)

XAML

Natürlich ist auch die Definition weiterer Absätze mit anderen Formatierungen möglich:

```
...
      <Paragraph FontSize="24">
        Hallo
        <Bold>User</Bold> , hier steht
        <Underline>jede Menge</Underline>
        <Italic>Text</Italic>
        <LineBreak/>
        <Hyperlink NavigateUri="http://www.doko-buch.de/">
            Die Autoren-Website ...
        </Hyperlink>
        <LineBreak />
      </Paragraph>
...
```

Ergebnis

HINWEIS: Hyperlinks sind in den hier vorgestellten WPF-Windows-Anwendungen nicht funktionsfähig, hier müssen Sie für die Logik selbst sorgen. Nutzen Sie dazu die Möglichkeit, die *Hyperlink*-Klasse abzuleiten und in der neuen Klasse das *RequestNavigate*-Ereignis für die Navigation zu verwenden.

HINWEIS: Formatierungen, wie Fett, Kursiv und Unterstrichen, können Sie in der *RichTextBox* zur Laufzeit problemlos per Tastatur-Befehl realisieren. Wählen Sie einfach *Strg+Shift+F*, *Strg+Shift+K* oder *Strg+Shift+U*[1].

[1] Die Tastenbelegung unterscheidet sich von der in der Hilfe angegebenen US-Belegung.

13.16.2 Neues Dokument zur Laufzeit erzeugen

Nachdem der Anwender mehr oder weniger ziellos mit dem Editor gearbeitet hat, kommt häufig der Wunsch auf, das Dokument komplett zu löschen.

Diese Aufgabe lösen Sie, indem Sie der *Document*-Eigenschaft einfach ein neues leeres *FlowDocument*-Objekt zuweisen.

BEISPIEL 13.47: Komplettes Dokument löschen bzw. erstellen

C#

```
private void newb_Click(object sender, RoutedEventArgs e)
{
    rtb1.Document = new FlowDocument();
}
```

13.16.3 Sichern von Dokumenten

Was nützt der beste Editor, wenn man den Inhalt nicht sichern kann? Eine direkte *Save*-Methode werden Sie sowohl am Control als auch beim *Document*-Objekt vergeblich suchen.

Gesichert werden kann das komplette Dokument (oder auch nur einzelne Abschnitte), indem Sie ein *TextRange*-Objekt aus dem Dokument erzeugen. Dazu müssen Sie den Beginn und das Ende des Textbereichs definieren. Der *TextRange* selbst bietet uns dann die gewünschte *Save*-Methode, bei der Sie auch das Datenformat (Text, RTF, XAML, XAML-Package) angeben können.

HINWEIS: Sichern können Sie in einen geöffneten Stream, damit stehen Ihnen alle Wege (Memory, Datenbank, Datei ...) offen.

BEISPIEL 13.48: Sichern des kompletten Inhalts des *RichtTextBox*

C#

```
private void Saveb_Click(object sender, RoutedEventArgs e)
{
```

Den nötigen Datei-Sichern-Dialog einblenden:

```
    SaveFileDialog dialog = new SaveFileDialog();
    dialog.Filter = "Xaml(*.xaml)|*.xaml";
```

Im Erfolgsfall:

```
    if ((bool)dialog.ShowDialog())
    {
```

Stream erzeugen:

```
        FileStream fs = (FileStream)dialog.OpenFile();
```

Das komplette Dokument als *TextRange* erfassen:

```
        TextRange tr = new TextRange(rtb1.Document.ContentStart,
                                     rtb1.Document.ContentEnd);
```

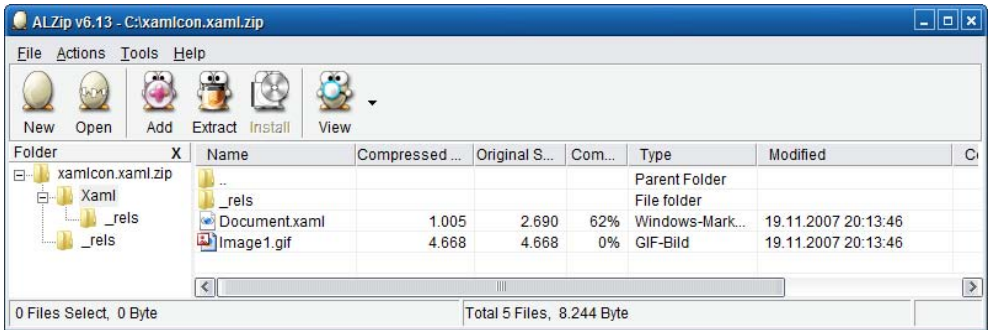
BEISPIEL 13.48: Sichern des kompletten Inhalts des *RichtTextBox*

Sichern:

```
                tr.Save(fs, DataFormats.XamlPackage);
                fs.Close();
            }
        }
```

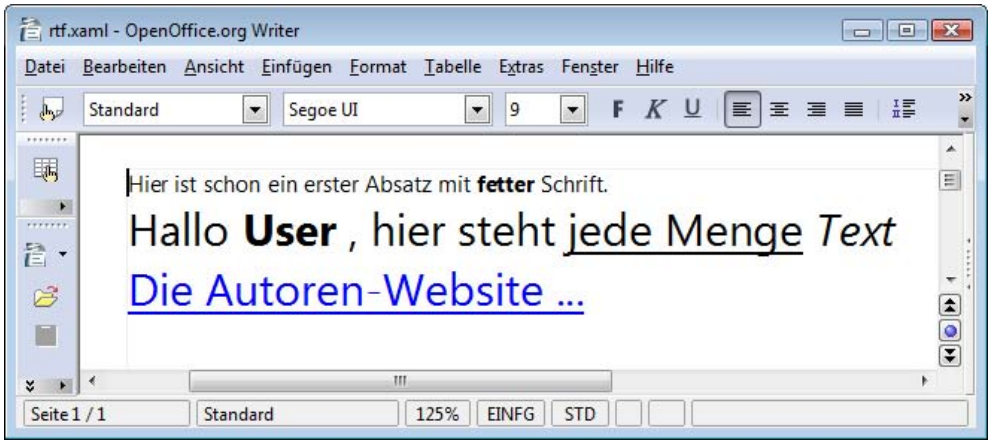
Bei der im Beispiel erzeugte XAML-Package-Datei handelt es sich um ein komprimiertes Format (ZIP), in dem sowohl die textuellen Inhalte als auch Grafiken etc. gesichert werden.

Möchten Sie den Inhalt dieser Dateien betrachten, genügt es, wenn Sie die Datei in .ZIP umbenennen und mit einem Entpacker-Programm öffnen:



Das Sichern im RTF-Format erfordert nur unwesentliche Änderungen an obigem Beispielcode. Sie müssen für den Dateidialog nur eine andere Extension vergeben und beim Aufruf der *Save*-Methode die Konstante *DataFormats.rtf* nutzen.

Dass der Export ganz gut funktioniert, sehen Sie an folgendem Beispiel, selbst der Hyperlink arbeitet wie gewünscht:



HINWEIS: Soll nur die aktuelle Auswahl gesichert werden, können Sie sich den Aufwand mit dem *TextRange*-Objekt sparen, die Eigenschaft *Selection* ist bereits ein *TextRange,* den Sie mit der Methode *Save* sichern können.

BEISPIEL 13.49: Aktuelle Auswahl sichern

C#

```
            FileStream fs = (FileStream)dialog.OpenFile();
            rtb1.Selection.Save(fs, DataFormats.XamlPackage);
            fs.Close();
```

13.16.4 Laden von Dokumenten

Soll das Dokument zu einem späteren Zeitpunkt wieder geladen werden, gehen wir im Grunde den selben Weg. Auch hier wird ein *TextRange*-Objekt erzeugt, in das wir diesmal den Inhalt laden.

BEISPIEL 13.50: Laden eines Dokuments (XAML-Package)

C#

```
        private void openb_Click(object sender, RoutedEventArgs e)
        {
```

Dateidialog öffnen:

```
            OpenFileDialog dialog = new OpenFileDialog();
            dialog.Filter = "Xaml(*.xaml)|*.xaml";
```

Im Erfolgsfall (Datei ausgewählt):

```
            if((bool)dialog.ShowDialog())
            {
```

Stream erzeugen und Bereich festlegen:

```
                FileStream fs = (FileStream)dialog.OpenFile();
                TextRange tr = new TextRange(rtb1.Document.ContentStart,
                                             rtb1.Document.ContentEnd);
```

Laden:

```
                tr.Load(fs, DataFormats.XamlPackage);
                fs.Close();
            }
        }
```

HINWEIS: Soll das geöffnete Dokument an der aktuellen Kursorposition eingefügt werden, nutzen Sie die *Selection*-Eigenschaft (siehe vorhergehendes Beispiel).

13.16.5 Texte per Code einfügen/modifizieren

Nicht alle Texte werden ausschließlich "von Hand" bearbeitet, viele Funktionen werden in Textverarbeitungen automatisiert. Und so stellt auch die *RichTextBox* reichlich Möglichkeiten zur codegesteuerten Bearbeitung der Texte zur Verfügung.

BEISPIEL 13.51: Einfügen des Tagesdatums an der aktuellen Kursor-Position

C# Die aktuelle Position des Kursors können Sie mit *Selection.End* ermitteln, diese Eigenschaft stellt ein *TextPointer*-Objekt zur Verfügung, über das sich unter anderem auch Texte einfügen lassen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    rtb1.Selection.End.InsertTextInRun(DateTime.Now.ToLongDateString());
}
```

BEISPIEL 13.52: Einfügen neuer Absatz

C# Auch hier nutzen wir wieder ein *TextPointer*-Objekt, um die Methode *InsertParagraphBreak* aufzurufen:

```
rtb1.Selection.End.InsertParagraphBreak();
```

BEISPIEL 13.53: Absatzfarbe ändern

C# Über ein *TextPointer*-Objekt haben Sie auch Zugriff auf den aktuellen Absatz, d h., Sie können dessen Eigenschaften bearbeiten:

```
rtb1.Selection.End.Paragraph.Background = Brushes.AliceBlue;
```

BEISPIEL 13.54: Neuen Absatz mit Fließtext am Dokumentende einfügen

C# Alle Absätze/Objekte werden über die *Blocks*-Collection verwaltet. Hängen Sie neue Absätze einfach an diese Auflistung an. Allerdings können Sie nicht wie in XAML direkt den Text im *Paragraph*-Objekt ausgeben, sondern Sie müssen einen Fließtext (*Run)* erzeugen und diesem den Text übergeben.

```
rtb1.Document.Blocks.Add(new Paragraph(new Run("Mein neuer Absatz")));
```

BEISPIEL 13.55: Text vor dem aktuellen Absatz einfügen

C# Auch hier hilft Ihnen die *Blocks*-Auflistung, allerdings müssen Sie in diesem Fall die *InsertBefore*-Methode aufrufen. Als Referenz ist die aktuelle Position des Textkursors wichtig. Die ermitteln wir, wie schon in den vorhergehenden Beispielen, über das *Selection*-Objekt.

```
rtb1.Document.Blocks.InsertBefore(rtb1.Selection.Start.Paragraph,
                                  new Paragraph(new Run("Neuer Absatz im Text")));
```

13.16.6 Texte formatieren

Sicher ist Ihnen auch schon aufgefallen, dass unsere *RichTextBox* einen entscheidenden Makel hat: es fehlt ein vernünftiger *ToolBar,* der uns zum Beispiel die bekannten Zeichenformatierungfunktionen zur Verfügung stellt.

BEISPIEL 13.56: *ToolBar*-Funktionalität für "Fett" und "Kursiv" realisieren.

XAML

Zunächst die Definition des *ToolBar*s, wir verwenden *ToggleButtons,* da es sich um einen Umschaltprozess handelt*:*

```
<Window x:Class="BSP_Controls.RichTextBox_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="RichTextBox_Bsp" Height="300" Width="481">

 <DockPanel>
   <ToolBarTray DockPanel.Dock="Top">
     <ToolBar >
```

Variante 1: Wir verwenden so genannte Commands, um die entsprechende Funktion (Fett/NichtFett) zu realisieren. Dazu muss die Schaltfläche mit dem Zielcontrol (*RichtTextBox)* über die Eigenschaft *CommandTarget* verbunden werden. Die eigentliche Funktion wird mit der Eigenschaft *Command* festgelegt:

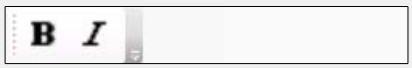
```
      <ToggleButton Name="boldb" Height="32" Width="32"
                    Command="EditingCommands.ToggleBold"
                    CommandTarget="{Binding ElementName=rtf1}">
        <Image Source="Images/boldhs.png" />
      </ToggleButton>
```

Variante 2: Wir nutzen das *Click*-Ereignis, um die Funktion zu realisieren:

```
      <ToggleButton Name="italicb" Height="32" Width="32" Click="italicb_Click">
        <Image Source="Images/italicHS.png" />
      </ToggleButton>
    </ToolBar>
  </ToolBarTray>
```

Ergebnis

Die erzeugte Werkzeugleiste:



C#

Das *Click*-Ereignis für die zweite Variante:

```
      private void italicb_Click(object sender, RoutedEventArgs e)
      {
```

Mit *ApplyPropertyValue* können Sie gezielt einzelne Eigenschaften des Textes beeinflussen. Übergeben wird der Name der Eigenschaft und der neue Wert:

BEISPIEL 13.56: *ToolBar*-Funktionalität für "Fett" und "Kursiv" realisieren.

```
            rtb1.Selection.ApplyPropertyValue(FlowDocument.FontStyleProperty,
                                              FontStyles.Italic);
        }
```

Starten Sie jetzt das Programm, führt ein Klick auf die *Kursiv*-Schaltfläche dazu, dass die entsprechende Formatierung auf den Text angewendet wird. Allerdings beschränkt sich der obige Code darauf, die Formatierung einmalig zu setzen. Ein Löschen ist so nicht möglich. Dazu müssten wir vorher den aktuellen Wert abfragen.

Alternativ bot sich die Variante mit der Verwendung von *Commands* an. Hier ist das Kommando "EditingCommands.ToggleBold", d h., jeder Klick führt zum Umschalten des bisherigen Zustands. Auf Quellcode können wir bei dieser Version gänzlich verzichten, damit dürfte diese Version der sinnvollste Weg sein.

Ein Problem haben wir jedoch nach wie vor, bewegen wir den Kursor durch die *RichTextBox,* werden die aktuellen Formatierungen nicht im ToolBar angezeigt. Über das *SelectionChanged*-Ereignis der *RichTextBox* können wir auf die Änderung der Kursorposition reagieren und den Status der Schaltflächen an die aktuelle Formatierung anpassen:

```
        private void rtb1_SelectionChanged(object sender, RoutedEventArgs e)
        {
```

Zunächst ermitteln wir den aktuellen Formatierungsstatus für Fett:

```
            object propval = rtb1.Selection.GetPropertyValue(FontWeightProperty);
```

Wenn die Eigenschaft bereits gesetzt ist,

```
            if (propval != DependencyProperty.UnsetValue)
```

können wir deren Wert auswerten:

```
                boldb.IsChecked = (FontWeight)propval == FontWeights.Bold;
```

Das Gleiche für Kursiv:

```
            propval = rtb1.Selection.GetPropertyValue(FontStyleProperty);
            if (propval != DependencyProperty.UnsetValue)
               italicb.IsChecked = (FontStyle)propval == FontStyles.Italic;
        }
```

HINWEIS: Obige Auswertung ist leider recht code-intensiv, da müssen sich die Microsoft-Programmierer sicherlich noch etwas Sinnvolleres einfallen lassen.

13.16.7 EditingCommands

Wie im vorhergehenden Beispiel gezeigt, ist mit der Verwendung von Commands eine recht einfache Möglichkeit gegeben, Aktoren (Schaltflächen etc.) mit Aktionen (*Command)* zu verknüpfen, die auf ein spezifisches Control wirken (*CommandTarget*).

HINWEIS: Mehr zu Commands im Abschnitt 14.4, wir wollen hier nicht zu viel vorgreifen.

Welche Commands Ihnen für die *RichTextBox* zur Verfügung stehen, soll die folgende Auflistung zeigen (die Bezeichner dürften selbsterklärend sein):

- **Cursorbewegung**
  (*Backspace, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord, MoveToLineEnd, MoveToLineStart, MoveToDocumentEnd, MoveToDocumentStart, MoveUpByLine, MoveUpByPage, MoveUpByParagraph, TabBackward, TabForward*)
- **Markierung**
  (*SelectDownByLine, SelectDownByPage, SelectDownByParagraph, SelectLeftByCharacter, SelectLeftByWord, SelectRightByCharacter, SelectRightByWord, SelectToDocumentEnd, SelectToDocumentStart, SelectToLineEnd, SelectToLineStart, SelectUpByLine, SelectUpByPage, SelectUpByParagraph*)
- **Schrift**
  (*DecreaseFontSize, IncreaseFontSize, ToggleBold, ToggleItalic, ToggleSubscript, ToggleSuperscript, ToggleUnderline*)
- **Absatzausrichtung**
  (*AlignCenter, AlignJustify, AlignLeft, AlignRight*)
- **Absatzformat**
  (*ToggleBullets, ToggleNumbering, DecreaseIndentation, IncreaseIndentation*)
- **Rechtschreibung**
  (*CorrectSpellingError, IgnoreSpellingError*)
- **Befehle**
  (*Delete, DeleteNextWord, DeletePreviousWord, EnterLineBreak, EnterParagraphBreak*)
- **Sonstiges**
  (*ToggleInsert*)

Alternativ sollten Sie auch einen Blick auf folgende Library werfen, diese bietet einen *RichTextBoxFormatBar*, der die wichtigsten Formatierungsoptionen anbietet:

LINK: `http://wpftoolkit.codeplex.com/`

13.16.8 Grafiken/Objekte einfügen

Dass neben Text auch andere Objekte in die *RichTextBox* eingefügt werden können, soll das folgende Beispiel zeigen, bei dem zur Laufzeit eine Grafik aus den Programmressourcen eingefügt und mit einem Ereignis verknüpft wird.

BEISPIEL 13.57: Grafik einfügen

C#

```
        private void imageb_Click(object sender, RoutedEventArgs e)
        {
```

Neues *Image*-Objekt erzeugen:

```
            Image img = new Image();
```

Die eigentliche Bitmap zuweisen:

```
            img.Source = BitmapFrame.Create(new
                                Uri("pack://application:,,,/images/frosch.gif"));
```

Größe bestimmen und Ereignis zuweisen:

```
            img.Height = 100;
            img.Width = 100;
            img.MouseDown += delegate {MessageBox.Show("Finger weg!");};
```

Und jetzt Grafik als neuen letzten Absatz einfügen:

```
            rtb1.Document.Blocks.Add(new Paragraph( new InlineUIContainer(img)));
        }
```

Controls wie *Image* etc. müssen in einem so genannten *InlineUIContainer* gekapselt werden, bevor sie im *Paragraph* eingefügt werden.

Ergebnis

Das Beispielprogramm in Aktion:

Sichern Sie jetzt das Dokument, wird die Grafik ebenfalls mit abgespeichert (z.B. in einem XAML-Package).

13.16.9 Rechtschreibkontrolle

Dass Sie die Rechtschreibkontrolle der *RichTextBox* mit

```
<RichTextBox Name="rtb1" SpellCheck.IsEnabled="True" ...
```

aktivieren können, dürfte Ihnen bereits von der einfachen *TextBox* her bekannt sein. Doch ein Blick in den Editor wird für lange Gesichert sorgen: alles Fehler und das trotz korrekter Schreibweise! Hier hilft das zusätzliche Setzen der Sprache:

```
<RichTextBox Name="rtb1" SpellCheck.IsEnabled="True" Language="de" ...
```

13.17 FlowDocumentPageViewer & Co.

Neben dem Editieren von formatierten Texten steht häufig auch deren Anzeige auf der Wunschliste des Programmierers. Neben der bereits vorgestellten *RichTextBox* bietet WPF hier mit

- *FlowDocumentPageViewer*,
- *FlowDocumentReader* und
- *FlowDocumentScrollViewer*

ein reichhaltiges Arsenal, um Flow-Dokumente anzuzeigen.

13.17.1 FlowDocumentPageViewer

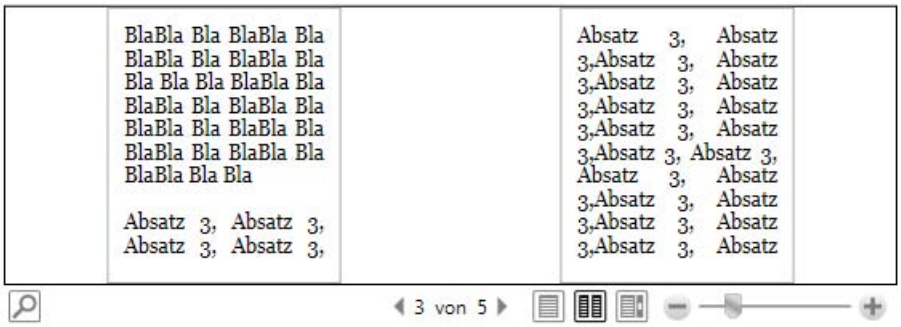
Für die seiten-orientierte Anzeige von Dokumenten bietet sich das *FlowDocumentPage-Viewer*-Control an. Über die Navigationstasten am unteren Rand können Sie zwischen den Seiten blättern, der Zoomfaktor lässt sich über einen Schieberegler vorgeben:

Hier ist schon ein erster Absatz mit fetter Schrift. Fülltext Fülltext Fülltext

13.17.2 FlowDocumentReader

Im Gegensatz zum vorhergehenden Control können Sie beim *FlowDocumentReader* zwischen verschiedenen Anzeigemodi wechseln (ein- und mehrseitig, sowie fortlaufend), die Textsuchfunktion steht per eigener Schaltfläche zur Verfügung.

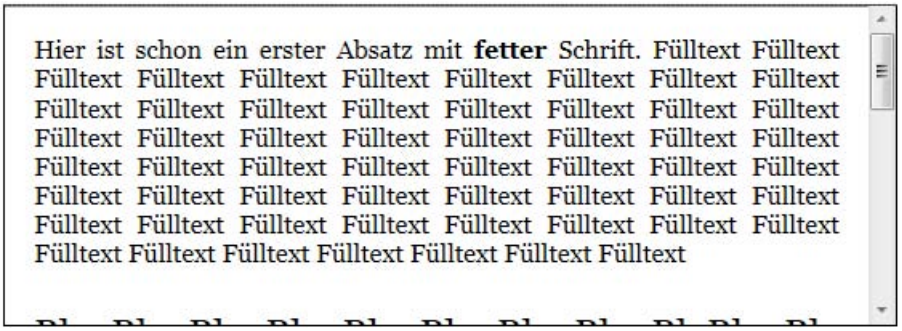
Zoom- und Blätter-Funktion entsprechen dem *FlowDocumentPageViewer*.



Durch die Unterstützung der verschiedenen Anzeigemodi ist dieser Viewer zwar leistungsfähiger, aber leider auch etwas langsamer.

13.17.3 FlowDocumentScrollViewer

Für die reine Fließtextdarstellung ohne Seiten sollten Sie den *FlowDocumentScrollViewer* einsetzen. Dieser verfügt lediglich über einen Scrollbar, um im Text zu blättern, der Zoomfaktor ist nur per Code (*Zoom*) einstellbar:



13.18 FlowDocument

Obwohl es eigentlich kein eigenes Control ist, wollen wir dennoch kurz auf das *FlowDocument* eingehen, verwenden wir dieses doch in der *RichTextBox* bzw. in den *FlowDocumentPageViewer, -Reader, -ScrollViewer*-Controls.

Ein *FlowDocument* stellt das Objektmodell eines formatierten Fließtextes dar, der in den o.g. Controls in unterschiedlichen Modi angezeigt oder auch editiert werden kann.

FlowDocument-Objekte können folgende untergeordneten Elemente enthalten:

- *Paragraph* (Absätze, die wiederum Texte, Floater, Figures etc. enthalten können)
- *BlockUIContainer* (Kapseln von Controls, z.B *Image*)
- *List* (eine Auflistung)
- *Section* (Zusammenfassung von Abschnitten)
- *Table* (eine Tabelle)

13.18.1 FlowDocument per XAML beschreiben

Zunächst wollen wir Ihnen an einem Beispiel die Beschreibung des *FlowDocuments* per XAML-Code demonstrieren, dieses können Sie direkt als *Content* einem der obigen Reader zuweisen:

BEISPIEL 13.58: Ein etwas umfangreicheres *FlowDocument*, das einige Möglichkeiten aufzeigt.

XAML

Zunächst das *FlowDocument* definieren (minimale Spaltenbreite, automatische Silbentrennung)

```
<FlowDocument ColumnWidth="400" IsHyphenationEnabled="True">
```

Für die folgenden Abschnitte gilt eine Schriftgröße von 12:

```
  <Section FontSize="12">
```

Ein erster Abschnitt mit Fließtext und verschiedenen Formatierungen:

```
    <Paragraph>
      <Bold>Fette Schrift</Bold> Hier steht Fließtext. Hier steht Fließtext.
      Hier steht Fließtext. <Underline>Hier steht Fließtext.</Underline> Hier steht
       Fließtext. Hier steht Fließtext.
      Hier steht Fließtext. Hier steht Fließtext. Hier steht Fließtext. Hier steht
Fließtext.
```

Ein frei positionierbarer Rahmen, in dem wiederum andere Elemente (in diesem Fall ein Absatz) enthalten sein können:

```
      <Figure Width="150" Height="100" Background="CornflowerBlue"
          HorizontalAnchor="PageLeft" HorizontalOffset="100" VerticalOffset="20">
        <Paragraph FontStyle="Italic" Foreground="White">
          Ein freier Bereich, der über Koordinatenangaben positioniert wird.
        </Paragraph>
      </Figure>
```

Auch dies ein Rahmen, der jedoch nicht absolut positioniert werden kann:

```
      <Floater Background="LightYellow" Width="300" HorizontalAlignment="Right">
        <Paragraph>
```

BEISPIEL 13.58: Ein etwas umfangreicheres *FlowDocument*, das einige Möglichkeiten aufzeigt.

```
            Noch ein freier Bereich, der horizontal über HorizontalAlignment positioniert
            werden kann.
          </Paragraph>
        </Floater>
      </Paragraph>
```

Ein normaler Absatz:

```
      <Paragraph>
        Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla Bla  ...
      </Paragraph>
```

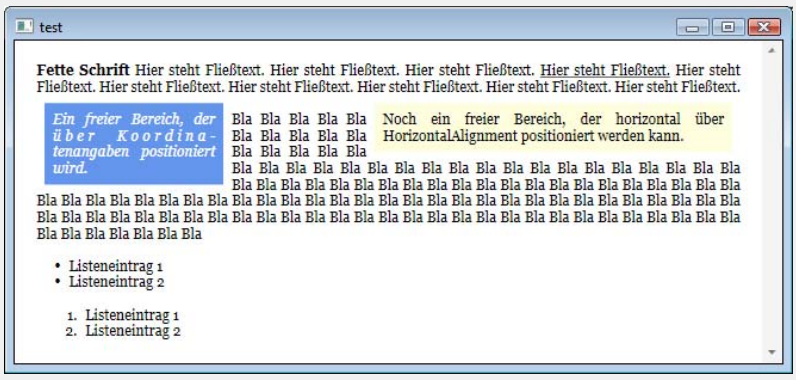
Eine Liste:

```
      <List MarkerStyle="Disc">
        <ListItem>
          <Paragraph>Listeneintrag 1</Paragraph>
        </ListItem>
        <ListItem>
          <Paragraph>Listeneintrag 2</Paragraph>
        </ListItem>
      </List>
```

Eine Liste mit Aufzählung:

```
      <List MarkerStyle="Decimal">
        <ListItem>
          <Paragraph>Listeneintrag 1</Paragraph>
        </ListItem>
        <ListItem>
          <Paragraph>Listeneintrag 2</Paragraph>
        </ListItem>
      </List>
...
    </Section>
  </FlowDocument>
```

Ergebnis



13.18.2 FlowDocument per Code erstellen

Neben der recht übersichtlichen Möglichkeit, Flow-Dokumente per XAML-Code zu erstellen (zum Beispiel auch direkt aus der *RichTextBox* heraus), können Sie natürlich auch Ihre Programmierfähigkeiten zum Einsatz bringen. Ein einfaches Beispiel zeigt die Vorgehensweise:

BEISPIEL 13.59: Ein *FlowDocument* per Code erstellen

C#

Fügen Sie in die Oberfläche zunächst einen *FlowDocumentPageViewer* ein und geben Sie diesem den Bezeichner "FlowDocumentPageViewer1".

Mit dem Laden des Fensters schreiten wir zur Tat:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
```

Eine *FlowDocument*-Instanz erzeugen:

```
    FlowDocument flowDoc = new FlowDocument();
```

Einen ersten Absatz hinzufügen, dieser enthält zunächst einen einfachen Text der fett ausgeben wird:

```
    Paragraph para = new Paragraph(new Bold(
                     new Run("Wir schreiben etwas Text in den ersten Absatz.")));
```

An den bestehenden Absatz hängen wir noch etwas Text an:

```
    para.Inlines.Add(new Run(" Hier kommt noch mehr Text im selben Absatz"));
```

Den Absatz an das *FlowDocument* anhängen:

```
    flowDoc.Blocks.Add(para);
```

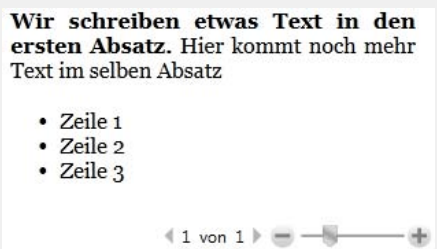
Eine Liste erzeugen:

```
    List liste = new List();
    liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 1"))));
    liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 2"))));
    liste.ListItems.Add(new ListItem(new Paragraph(new Run("Zeile 3"))));
    flowDoc.Blocks.Add(liste);
```

Das *FlowDocument* zur Anzeige bringen:

```
    FlowDocumentPageViewer1.Document = flowDoc;
}
```

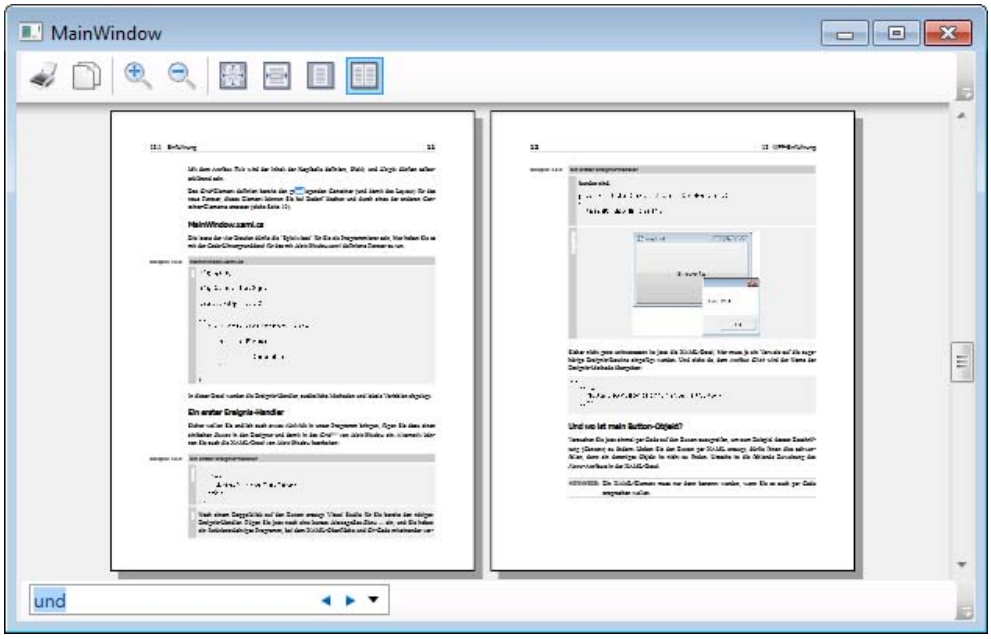
Ergebnis



HINWEIS: Auf weitere Möglichkeiten (Controls, Tabellen etc.) von Flow-Dokumenten wollen wir an dieser Stelle nicht eingehen, das überlassen wir besser der Spezialliteratur.

13.19 DocumentViewer

Neben den auf die Flow-Dokumente festgelegten Controls findet sich auch ein *DocumentViewer*-Control, das ausschließlich zur Anzeige von XPS-Dokumenten verwendet werden kann. Neben einer Druckoption können Sie die Daten auch in die Zwischenablage kopieren, die Ansicht skalieren und zwischen unterschiedlichen Seitendarstellungen wechseln. Last but not least, verfügt das Control auch über ein einfache Suchfunktion innerhalb des Textes.



Wie Sie ein externes XPS-Dokument laden, zeigt das folgende Beispiel:

BEISPIEL 13.60: Laden eines XPS-Dokuments in den *DocumentViewer*

XAML

```
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="MainWindow" Height="238" Width="348" Loaded="Window_Loaded"
        WindowStartupLocation="CenterScreen">
        <DocumentViewer Name="DocumentViewer1" />
</Window>
```

BEISPIEL 13.60: Laden eines XPS-Dokuments in den *DocumentViewer*

C#

Fügen Sie zunächst die Assembly *ReachFramework.dll* als Verweis zu Ihrem Projekt hinzu. Nachfolgend können Sie den entsprechenden Ereigniscode übernehmen.

Namespaces importieren:

```
...
using System.Windows.Xps.Packaging;
using System.IO;

namespace WpfApplication2
{
    public partial class MainWindow : Window
    {
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Zunächst ein *XpsDocument* aus der Datei "c:\Test.xps" erstellen, nachfolgend können wir eine *FixedDocumentSequence* für die Anzeige abrufen:

```
            XpsDocument doc = new XpsDocument("c:\\test.xps", FileAccess.Read);
            DocumentViewer1.Document = doc.GetFixedDocumentSequence();
        }
```

HINWEIS: Mehr zu diesem Thema finden Sie im Abschnitt 16.1.1 "XPS-Dokumente", wo wir uns sowohl dem Erstellen von XPS-Dokumenten als auch der Verwendung der *DocumentViewer*-Komponente zuwenden werden.

13.20 Expander, TabControl

Gerade Dialogfenster mit vielen Eingabefeldern/Optionen leiden unter immer demselben Problem: es ist zu wenig Platz vorhanden. Diesem Missstand sollen das *Expander*- und das *TabControl* abhelfen.

13.20.1 Expander

Das *Expander*-Control ermöglicht es, zwischen einem aufgeklappten und einem geschlossenen Zustand hin- und herzuschalten. Die Höhe des Controls im aufgeklappten Zustand bestimmt sich aus der Höhe des enthaltenen Layout-Containers bzw. des enthaltenen Controls.

Sichtbar bleibt in jedem Fall die per *Header* definierte Beschriftung und eine Schaltfläche, mit der Sie das Control aufklappen können. Die Aufklapprichtung wird mit *ExpandedDirection* (*Down, Left, Right, Up*) definiert.

BEISPIEL 13.61: Verwendung Expander

XAML

Zunächst ordnen wir alle drei *Expander* in einem *StackPanel* an:

```
<StackPanel>
```

Die Beschriftung, Hintergrundfarbe und die Aufklapprichtung festlegen:

```
  <Expander Header="Private Daten" Background="LightGray" ExpandDirection="Right" >
```

Der Inhalt des Expanders ist z.B. ein weiteres *StackPanel*, mit dem die Eingabefelder beschriftet bzw. angeordnet werden:

```
    <StackPanel Margin="5">
      <Label Content="Vorname:" />
      <TextBox></TextBox>
      <Label Content="Nachname:"/>
      <TextBox></TextBox>
```

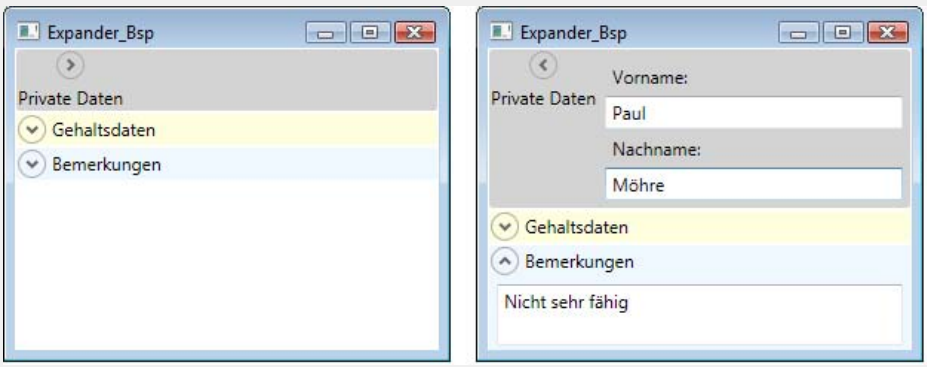
Die Höhe des Expanders wird durch die Summe der einzelnen Controls bestimmt:

```
    </StackPanel>
  </Expander>
  <Expander Header="Gehaltsdaten" Background="LightYellow">
    <StackPanel Margin="5">
      <Label Content="Gehalt:" />
      <TextBox></TextBox>
      <Label Content="Steuerklasse:"/>
      <TextBox></TextBox>
    </StackPanel>
  </Expander>
```

Hier blenden wir statt eines Layout-Controls gleich eine *TextBox* im *Expander* ein und legen die Höhe der *TextBox* entsprechend der gewünschten Höhe fest:

```
  <Expander Header="Bemerkungen" Background="AliceBlue">
    <TextBox Margin="5" Height="150">
    </TextBox>
  </Expander>
</StackPanel>
```

Ergebnis



HINWEIS: Über die *IsExpanded*-Eigenschaft können Sie per Code den aktuellen Zustand abfragen bzw. beeinflussen (z.B. Schließen bei Fokusverlust).

13.20.2 TabControl

Statt wie beim *Expander* einzelne Bereich einzublenden, werden beim *TabControl* die einzelnen Registerkarten (*TabItem*s) komplett umgeschaltet, es ist also immer nur eine Seite sichtbar. Wo das Register angezeigt wird, bestimmen Sie mit der Eigenschaft *TabStripPlacement* (*Left*, *Right*, *Top*, *Bottom*). Die Auswahl des aktiven *TabItem*s erfolgt mit der Maus oder über die Eigenschaften *SelectedIndex* bzw. *SelectedItem*. Alternativ können Sie mit *IsSelected* den Status eines *TabItem*s abfragen.

BEISPIEL 13.62: Ein *TabControl* mit drei *TabItems*

XAML

```
<Window x:Class="BSP_Controls.Tab_Bsp" ...>
```

Das *TabControl*:

```
 <TabControl>
```

Und hier auch schon die erste Registerkarte mit einfacher Beschriftung:

```
  <TabItem Header="Tabseite 1">
    Hier steht der Content!
  </TabItem>
```

Die zweite Registerkarte mit einem Layout-Control im Content:

```
  <TabItem Header="Tabseite 2">
    <StackPanel>
      <Label>Beliebige Controls ...</Label>
    </StackPanel>
  </TabItem>
```

Die dritte Registerkarte macht von der Möglichkeit Gebrauch, im Header auch Grafiken bzw. beliebige Controls anzuzeigen:

```
  <TabItem>
    <TabItem.Header>
      <StackPanel Orientation="Horizontal">
        <Image Source="Images/rudi.gif" Height="30"/>
        <TextBlock  VerticalAlignment="Center">Tabseite 3</TextBlock>
      </StackPanel>
    </TabItem.Header>
```

Der Content besteht wiederum aus einem Layout-Control und den darin enthaltenen Controls:

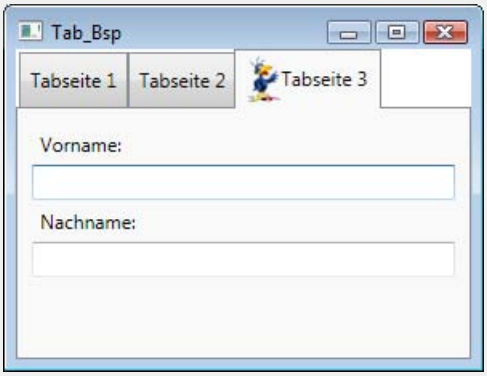
```
    <StackPanel Margin="5">
      <Label Content="Vorname:" />
      <TextBox></TextBox>
      <Label Content="Nachname:"/>
      <TextBox></TextBox>
```

BEISPIEL 13.62: Ein *TabControl* mit drei *TabItems*

```
    </StackPanel>
  </TabItem>
 </TabControl>
</Window>
```

Ergebnis

Das *TabControl* zur Laufzeit:



HINWEIS: Möchten Sie per Programm auf Änderungen reagieren, nutzen Sie das *SelectionChanged*-Ereignis für die Auswertung.

13.21 Popup

Für die Anzeige von Zusatzinformationen können Sie den *ToolTip* eines Controls verwenden. Dieser hat allerdings den Nachteil, dass er einem spezifischen Control zugeordnet ist. Anders das Popup-Fenster (ja es handelt sich um ein eigenes Fenster!), das zwar nicht automatisch eingeblendet wird, dafür aber an jeder beliebigen Stelle stehen kann.

HINWEIS: Einmal eingeblendet, bleibt das Fenster an der aktuellen Position stehen, auch wenn das übergeordnete Fenster verschoben wird.

BEISPIEL 13.63: Einfaches *Popup* einblenden

XAML

```
...
 <Canvas>
   <Button Canvas.Left="206" Canvas.Top="37" Height="30" Name="button1" Width="43"
           Click="button1_Click">...</Button>
   <Popup Name="pop2" Placement="MousePoint" VerticalOffset="25" HorizontalOffset="25"
          Width="100">
     <TextBlock Height="50" Margin="1" TextWrapping="Wrap" Background="LightBlue" >
        Bla Bla Bla ...
```

BEISPIEL 13.63: Einfaches *Popup* einblenden

```
    </TextBlock>
  </Popup>
```

C#

Das Ein-/Ausblenden des Popup-Fensters wird mit der Eigenschaft *IsOpen* realisiert:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    pop2.IsOpen = !pop2.IsOpen;
}
```

Ergebnis

Klicken Sie zur Laufzeit auf den Button, wird das Popup-Fenster relativ zur aktuellen Cursorpostion (siehe *Placement*) eingeblendet:

Weitere Möglichkeiten zum Platzieren bieten sich mit den folgenden Eigenschaften:

- *Placement* (*Absolute, Bottom, Mouse, Relative ...*)
- *PlacementTarget* (ein Control, auf das sich relative Angaben beziehen)
- *PlacementRectangle* (optionales Rechteck, zu dem das Popup relativ angezeigt wird)
- *HorizontalOffset* (zusätzliche horizontale Verschiebung der Koodinaten)
- *VerticalOffset (*zusätzliche vertikale Verschiebung der Koordinaten)

Neben der Positionierung bietet sich auch die Möglichkeit, das Popup-Fenster animiert einzublenden. Setzen Sie dazu die Eigenschaft *PopupAnimation* auf *Fade*, *Scroll* oder *Slide.*

HINWEIS: Damit die Animation auch sichtbar ist, muss *AllowsTransparency* auf *True* gesetzt werden.

BEISPIEL 13.64: Weitere Möglichkeiten für die Konfiguration des *Popup*

XAML

Relative Platzierung zu einer *TextBox*, automatisches Einblenden mit Animation (wenn die *TextBox* den Fokus erhält) bzw. Ausblenden (wenn Fokusverlust).

```
...
  <TextBox Canvas.Left="28" Canvas.Top="37" Height="31" Name="textBox1" Width="168"/>
  <Popup Name="pop1" Width="180"
```

Das Bezugselement für die Koordinatenangaben festlegen:

```
        PlacementTarget="{Binding ElementName=textBox1}" Placement="Relative"
```

BEISPIEL 13.64: Weitere Möglichkeiten für die Konfiguration des *Popup*

Die relative Platzierung:

```
VerticalOffset="35" HorizontalOffset="130"
```

Die Animation festlegen:

```
PopupAnimation="Slide" AllowsTransparency="True"
```

Mittels Datenbindung an die *IsFocused*-Eigenschaft der *TextBox* öffnen/schließen wir das Popup-Fenster:

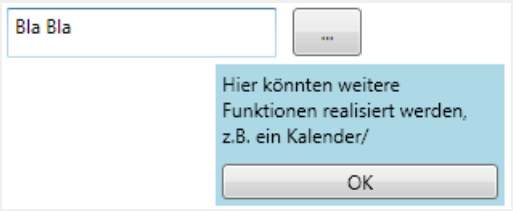
```
IsOpen="{Binding ElementName=textBox1, Path=IsFocused, Mode=OneWay}">
```

Hier folgen die Inhalte des Popup-Fensters:

```
<StackPanel Background="LightBlue">
  <TextBlock Height="50" Margin="4" TextWrapping="Wrap" Background="LightBlue" >
      Hier könnten weitere Funktionen realisiert werden, z.B. ein
      Kalender/Taschenrechner
  </TextBlock>
  <Button Margin="4">OK</Button>
</StackPanel>
</Popup>
```

Ergebnis

Nachdem die *TextBox* den Fokus erhalten hat, dürfte das Popup-Fenster eingeblendet werden:



13.22 TreeView

Sicher jedem vom Explorer her bekannt, darf auch in WPF ein *TreeView*-Control nicht fehlen. Allerdings könnte die Verwendung dieses Controls viele gestandene Windows-Forms und Win32-Programmierer schnell in den Wahnsinn treiben, unterscheidet sich doch die Programmierung in vielen Punkten vom bisherigen Vorgehen. Dies ist vor allem der freien Programmierbarkeit dieses Controls geschuldet. Was Sie in den einzelnen *TreeViewItems* anzeigen ist, wie in WPF üblich, von Ihnen frei definierbar.

Doch ganz so kompliziert wollen wir nicht anfangen, ein einfaches Beispiel zeigt zunächst, wie Sie eine rein textorientierte *TreeView* per XAML-Code definieren:

BEISPIEL 13.65: Grundprinzip der Schachtelung der *TreeViewItem*-Elemente

XAML

Definition der eigentlichen *TreeView*:

```
<TreeView>
```

Der erste Knoten, über die *Header*-Eigenschaft bestimmen Sie die Beschriftung:

```
  <TreeViewItem Header ="Root" IsExpanded="True">
```

Unter-Knoten schachteln Sie einfach:

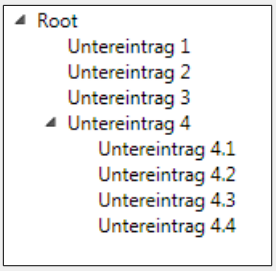
```
    <TreeViewItem Header ="Untereintrag 1" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 2" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 3" IsExpanded="True"/>
    <TreeViewItem Header ="Untereintrag 4" IsExpanded="True">
```

Eine weitere Unterebene:

```
      <TreeViewItem Header ="Untereintrag 4.1" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.2" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.3" IsExpanded="True"/>
      <TreeViewItem Header ="Untereintrag 4.4" IsExpanded="True"/>
    </TreeViewItem>
  </TreeViewItem>
</TreeView>
```

Ergebnis

Die Darstellung der so definierten *TreeView*:



BEISPIEL 13.66: Alternativ können Sie die *TreeView* auch per Code füllen:

C#

```
        private void Btn1_Click(object sender, RoutedEventArgs e)
        {
            TreeViewItem tvi;
```

Bisherige Inhalte löschen:

```
            Tv1.Items.Clear();
```

Den ersten Eintrag erzeugen und weitere Untereinträge hinzufügen:

```
            tvi = new TreeViewItem { Header = "Root" };
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 1" });
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 2" });
```

BEISPIEL 13.66: Alternativ können Sie die *TreeView* auch per Code füllen:

```
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 3" });
            tvi.Items.Add(new TreeViewItem { Header = "Untereintrag 4" });
            Tv1.Items.Add(tvi);
        }
```

Was ist daran so kompliziert, werden Sie sicher nach den bisherigen Beispielen fragen. Die Antwort kommt spätestens bei der Aufgabenstellung, Grafiken in die Einträge einzufügen.

BEISPIEL 13.67: Frei definierte *TreeViewItems* mit Grafiken bzw. mit Schaltfläche

XAML

```
<TreeView>
```

Hier nochmal ein ganz "normaler" Knoten:

```
  <TreeViewItem Header ="Root" IsExpanded="True" Tag="0">
```

Und jetzt wird es kompliziert:

```
    <TreeViewItem>
```

Eine Image-Eigenschafte gibt es nicht, stattdessen erzeugen wir den kompletten Eintrag per *StackPanel*, das wiederum aus einem *Image* und einem *TextBlock* besteht:

```
      <TreeViewItem.Header>
        <StackPanel Orientation="Horizontal">
          <Image Source="images\frosch.gif" Height="40"/>
          <TextBlock Text="Ein Frosch" VerticalAlignment="Center"/>
        </StackPanel>
      </TreeViewItem.Header>
    </TreeViewItem>
    <TreeViewItem>
      <TreeViewItem.Header>
        <StackPanel Orientation="Horizontal">
          <Image Source="images\rudi.gif" Height="40"/>
          <TextBlock Text="Ein Rabe" VerticalAlignment="Center"/>
        </StackPanel>
      </TreeViewItem.Header>
    </TreeViewItem>
```

Dass es auch mit ganz beliebigen Controls geht, zeigt die folgende Variante, bei der wir einen *Button* im *TreeViewItem* einblenden:

```
    <TreeViewItem>
      <TreeViewItem.Header>
        <Button>Eine Schaltfläche</Button>
      </TreeViewItem.Header>
    </TreeViewItem>
  </TreeViewItem>
</TreeView>
```

BEISPIEL 13.67: Frei definierte *TreeViewItems* mit Grafiken bzw. mit Schaltfläche

Ergebnis

Nach all den "Oberflächlichkeiten" wollen wir uns jetzt den "inneren Werten" des Controls zuwenden:

- Den aktuellen Knotenzustand können Sie mit *IsExpanded* abfragen oder setzen.
- Der aktuell gewählte Eintrag lässt sich über die Eigenschaft *SelectedItem* ermitteln, alternativ können Sie auch die Liste der Einträge durchlaufen und die *IsSelected*-Eigenschaft abfragen.
- Auf Änderungen der Auswahl können Sie mit *SelectedItemChanged* bzw. *Selected* (für *TreeViewItem*) reagieren.
- Um bei Bedarf Scrollbars einzublenden, sollten Sie *Width* und *Height* der *TreeView* explizit setzen.
- Mit der Methode *BringIntoView* können Sie gezielt einen *TreeViewItem* einblenden, d h., bei Bedarf werden Knoten geöffnet und die *TreeView* scrollt den gewünschten Eintrag in den sichtbaren Bereich.
- Nutzen Sie die *Tag*-Eigenschaft, um einzelnen *TreeViewItem*s zusätzliche Informationen zuzuordnen, oder erzeugen Sie gleich eine neue Klasse, die Sie von *TreeViewItem* ableiten und um zusätzliche Eigenschaften und Methoden bereichern.

HINWEIS: Um der Thematik "DataBinding" in Kapitel 15 nicht vorzugreifen, verzichten wir an dieser Stelle auf weitere Erläuterungen.

13.23 ListView

Die *ListView* als Ableitung der schon vorgestellten *ListBox* zeigt zunächst das gleiche Verhalten, wenn Sie einem *ListView*-Element weitere *ListViewItems* hinzufügen.

BEISPIEL 13.68: *ListView* mit *ListViewItems*

XAML

```
<ListView>
  <ListViewItem>Zeile 1</ListViewItem>
  <ListViewItem>Zeile 2</ListViewItem>
  <ListViewItem>Zeile 3</ListViewItem>
</ListView>
```

BEISPIEL 13.68: *ListView* mit *ListViewItems*

Ergebnis

Die erzeugte Ansicht dürfte Sie nicht überraschen:

Doch im Gegensatz zur einfachen *ListBox* lassen sich für einen *ListView* so genannte Ansichten (Views) definieren, die auch recht einfach austauschbar sind. Eine dieser Ansichten ist die von WPF bereits vordefinierte *GridView,* die eine Collection in Tabellenform darstellen kann.

HINWEIS: Die einzelnen Elemente innerhalb diese Ansicht werden nicht per XAML/Code, sondern per Datenbindung zugeordnet.

BEISPIEL 13.69: Prinzip der Datenbindung in einem *ListView*-Control

XAML

Zunächst die Datenquelle zuordnen:

```
<ListView Name="lv1" ItemTemplate="{DynamicResource CustomerTemplate}"
          ItemsSource="{Binding Path=Table}">
```

Hier wird die spezielle View, in diesem Fall die *GridView*, definiert:

```
  <ListView.View>
    <GridView>
```

Die Spalten der *GridView* mit den Spalten der obigen Datenquelle verbinden:

```
      <GridViewColumn Header="Id" DisplayMemberBinding="{Binding Path=Id}"/>
      <GridViewColumn Header="Vorname" DisplayMemberBinding="{Binding Path=VName}"/>
      <GridViewColumn Header="Nachname" DisplayMemberBinding="{Binding Path=NName}"/>
    </GridView>
  </ListView.View>
</ListView>
```

C#

Per Code müssen Sie zur Laufzeit noch die *Table* eines *DataSet*s zuweisen:

```
lv1.DataContext = ds.Tables[0].DefaultView;
```

13.24 DataGrid

Welcher Programmierer hat es bisher nicht vermisst? Für viele war das Fehlen eines leistungsfähigen *DataGrid*s, das im Gegensatz zur *ListView* auch das Editieren der Daten beherrscht, ein entscheidender Faktor für das Festhalten an "alten" Windows Forms-Anwendungen. Doch das Warten hat sich gelohnt, das früher nur im zusätzlichen WPF-Toolkit enthaltene *DataGrid* findet sich seit der Framework-Version 4.0 auch im WPF-Werkzeugkasten.

HINWEIS: Da das Control mit einem großem Funktionsumfang aufwarten kann, beschränken wir uns im Rahmen dieses Buchs auf einige praktische Aufgabenstellungen, die wir aber aus naheliegenden Gründen im Kapitel 15 ab Seite 825 untergebracht haben.

13.25 Calendar/DatePicker

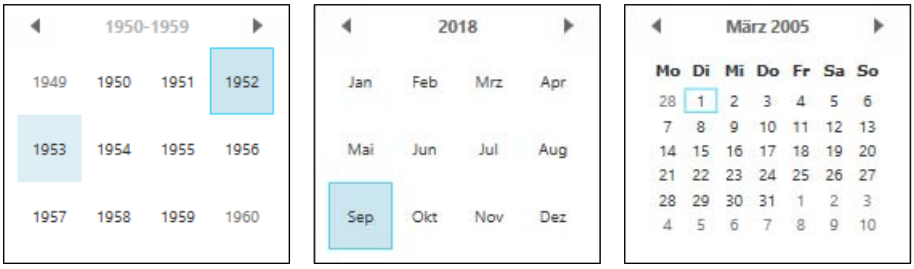
Seit WPF 4 gibt es auch zwei Controls für die Anzeige/Eingabe von Kalenderdaten. Während der *Calendar* für die Anzeige von Jahres- bzw. Monatsübersichten geeignet ist, steht der *DatePicker* lediglich für die Eingabe eines Datums zur Verfügung.

Die beiden Controls:

Einige Anwendungsbeispiele zeigen Ihnen die Verwendung.

DisplayMode

Wie schon erwähnt, lässt das *Calendar*-Control über die *DisplayMode*-Eigenschaft unterschiedliche Anzeigemodi (*Decade, Year, Month*) zu:



DisplayDate, SelectedDate und SelectedDates

Diese wohl wichtigsten Eigenschaften geben Auskunft über die aktuelle Auswahl im jeweiligen Control:

- Bei *DisplayDate* handelt es sich um den Anzeigewert des Controls nach dem Start. Dies ist meist das aktuelle Datum.

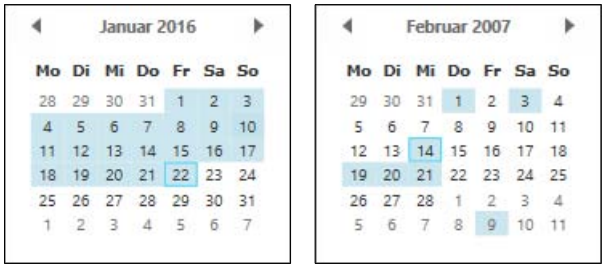
- *SelectedDate* entspricht der Benutzerauswahl, es handelt sich um **einen** Datumswert (Voraussetzung ist der *SingleDate*-Auswahlmodus).
- Die *SelectedDates*-Collection können Sie nutzen, wenn das *Calendar*-Control eine Mehrfachauswahl zulässt.

Auswahlmodi

Für den *DatePicker* steht die Frage nicht, hier können Sie immer nur ein einziges Datum wählen, wohingegen der *Calendar* mittels *SelectionMode*-Eigenschaft zwischen folgenden Modi unterscheiden kann:

- *None* (nur Anzeige von *DisplayDate*)
- *SingleDate* (Einzeldatum; per *SelectedDate* auslesen)
- *SingleRange* (Datumsbereich; per *SelectedDates* auslesen) und
- *MultipleRange* (mehrere Datumsbereiche; per *SelectedDates* auslesen)

SingleRange: *MultipleRange:*



Die Auswahl der Bereiche per XAML-Code erfordert etwas Schreibarbeit:

BEISPIEL 13.70: Bereichsauswahl per XAML-Code

XAML

```
<Window x:Class="BSP_Controls.Calendar_Bsp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        Title="Calendar_Bsp" Height="300" Width="300">
    <StackPanel>
        <Calendar Name="Calendar1" DisplayMode="Month" SelectionMode="MultipleRange">
            <Calendar.SelectedDates>
                <sys:DateTime>2/15/2020</sys:DateTime>
                <sys:DateTime>2/16/2020</sys:DateTime>
                <sys:DateTime>2/18/2020</sys:DateTime>
            </Calendar.SelectedDates>
        </Calendar>
```

BEISPIEL 13.70: Bereichsauswahl per XAML-Code

Ergebnis

Der Nutzer hat jetzt die Möglichkeit, diese Bereiche zu belassen oder zu erweitern. Die Abfrage der Bereiche erfolgt per *SelectedDates*-Collection:

BEISPIEL 13.71: Abfrage der gewählten Datumswerte

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    foreach (var datum in Calendar1.SelectedDates)
       MessageBox.Show(datum.ToString());
}
```

Sperrtage

Zu einem guten Kalender gehört auch die Möglichkeit, einzelne Datumswerte zu sperren. Über die Collection *BlackoutDates* steht Ihnen diese Funktionalität zur Verfügung.

HINWEIS: In diesem Fall müssen Sie *DateRange*-Eigenschaften setzen (von/bis).

BEISPIEL 13.72: Auswahl von Sperrtagen

XAML

```
<Window x:Class="BSP_Controls.Calendar_Bsp"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:sys="clr-namespace:System;assembly=mscorlib"
       Title="Calendar_Bsp" Height="300" Width="300">
   <StackPanel>
       <Calendar Name="Calendar1" DisplayMode="Month" SelectionMode="MultipleRange">
```

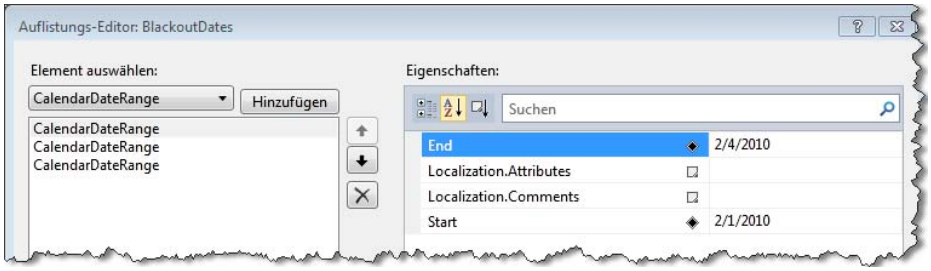
Setzen der Bereiche:

```
           <Calendar.BlackoutDates>
              <CalendarDateRange Start="2/1/2010" End="2/4/2010"/>
              <CalendarDateRange Start="2/11/2010" End="2/12/2010"/>
              <CalendarDateRange Start="2/16/2010" End="2/16/2010"/>
           </Calendar.BlackoutDates>
       </Calendar>
```

BEISPIEL 13.72: Auswahl von Sperrtagen

Ergebnis

Alternativ steht Ihnen ein entsprechender Auflistungseditor zur Verfügung:



Calendar skalieren

Wollen Sie das *Calendar*-Control skalieren (*Width/Height*) werden Sie sicher zunächst enttäuscht sein, das Control ändert seine Größe um keinen Millimeter (bzw. Pixel)! Doch nicht verzagen, mit zwei Alternativen bekommen Sie das Control auf die gewünschte Größe:

BEISPIEL 13.73: Skalieren mit der *ViewBox*

XAML

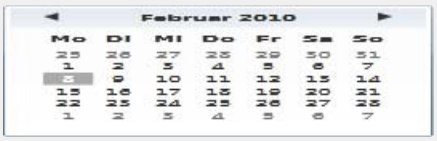
```
<Viewbox>
    <Calendar x:Name="CalendarControl"
              HorizontalAlignment="Left" VerticalAlignment="Top">
    </Calendar>
</Viewbox>
```

BEISPIEL 13.74: Skalieren mit Transformation

XAML

```
<Calendar  HorizontalAlignment="Left" VerticalAlignment="Top">
    <Calendar.RenderTransform>
        <ScaleTransform ScaleX="1.5" ScaleY=".5" />
    </Calendar.RenderTransform>
</Calendar>
```

Ergebnis



HINWEIS: Weitere Anpassungsmöglichkeiten für das Control bestehen durch die Verwendung von Templates, wir gehen ab Seite 754 darauf ein.

Damit wollen wir die Thematik "Kalender" abschließen.

13.26 InkCanvas

Mit dem *InkCanvas* möchten wir Ihnen noch ein zunächst recht unscheinbares Control vorstellen, das im Zuge der größer werdenden Verbreitung von Tablet-PCs sicher noch an Bedeutung gewinnen wird. Das Control stellt eine Zeichenfläche zur Verfügung, in der Sie mit einem Stift (oder auch der Maus) freie Zeichnungen realisieren oder Markierungen vornehmen können. Dazu kann optional in den Hintergrund des Controls eine Grafik eingeblendet werden (z.B. ein Wegeplan).

HINWEIS: Auf das Thema "Multitouch" (ab Windows 7) gehen wir im Rahmen dieses Buchs nicht ein, es mangelt den Autoren schlicht an der nötigen Hardware.

Die von Maus, Stift oder Code erzeugten Linien/Striche werden in einer internen *Strokes*-Collection verwaltet. Einzelne *Stroke*-Objekte bestehen wiederum aus Zeichenpunkten (*StylusPoints*), haben eine Stiftform (*StylusTip),* eine Größe (*Width, Height*) und weisen eine Farbe auf.

Die Größe des Controls richtet sich beim Entwurf zunächst nach dem umgebenden Layout-Control. Handelt es sich bei diesem zum Beispiel um einen *ScrollViewer,* wird die Größe des *InkCanvas* zur Laufzeit an die maximalen Stiftpositionen angepasst. Das heißt, zeichnen Sie über den "Rand" des Controls hinweg, werden die Stiftbewegungen zunächst aufgezeichnet. Erst nach dem Zeichenende werden die Abmessungen des Controls an die nun neuen maximalen Ausdehnungen angepasst.

13.26.1 Stift-Parameter definieren

Wie schon erwähnt, kann der Zeichenstift verschiedene Eigenschaften aufweisen, die zusammen mit den Eingabekoordinaten in der *Strokes*-Collection abgespeichert werden.

Einstellen können Sie diese Werte über die *DefaultDrawingAttributes*-Eigenschaft des *InkCanvas*:

Eigenschaft	Beschreibung
Color	Die für den Stift verwendete Farbe.
FitToCurve	Sollen die Linien geglättet werden[1], setzen Sie den Wert auf *True*. Nach der Zeichenoperation wird in diesem Fall die Linie automatisch an einen Kurvenverlauf angepasst.
Height, Width	Bestimmt die Höhe und Breite des Zeichenstifts.
StylusTip	Bestimmt die Grundform des Zeichenstifts (Rectangle, Ellipse).

[1] z B. um Zitterbewegungen und Ruckler auszugleichen ...

BEISPIEL 13.75: Linienverlauf ohne und mit *FitToCurve*

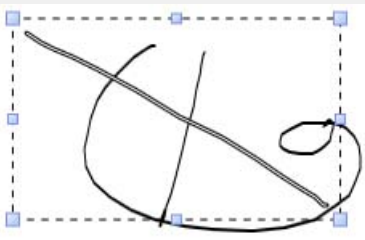
BEISPIEL 13.76: Linie mit rundem und eckigem Stift

13.26.2 Die Zeichenmodi

Der *InkCanvas* kennt verschiedene Zeichenmodi (*EditingMode*), über die Sie die jeweils ausführbare Aktion vorgeben:

EditingMode	Beschreibung
None	Keine Reaktion auf Stifteingaben.
Ink	Normales Zeichnen.
GestureOnly	Während der Stiftbewegung ist die Figur sichtbar, diese wird nach dem Zeichenende gelöscht. Dieser Modus dient der Auswertung der Gestik (z.B. ein gezeichneter Kreis, eine Linie etc.).
InkAndGesture	Zeichnen und Gestikauswertung.
Select	Auswahl von *Stroke*-Objekten, diese können zum Beispiel verschoben oder gelöscht werden.
EraseByPoint	Löschen von Linienstücken (ähnlich Radiergummi in einem Malprogramm).
EraseByStroke	Löschen kompletter *Stroke*-Objekte (ähnlich einem Vektorgrafikprogramm).

BEISPIEL 13.77: Markierte (Select) *Stroke*-Objekte



13.26.3 Inhalte laden und sichern

Möchten Sie die Zeichnung sichern bzw. zu einem späteren Zeitpunkt erneut laden, müssen Sie sich mit der *Strokes*-Collection beschäftigen. Diese verfügt über die erforderliche *Save*-Methode bzw. einen geeigneten Konstruktor, um die Daten aus einem Stream einzulesen.

BEISPIEL 13.78: Sichern der Daten

C#
```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    using (FileStream fs = new FileStream(@"c:\test.ink", FileMode.Create,
                                          FileAccess.Write))
    {
        InkCanvas1.Strokes.Save(fs);
        fs.Close();
    }
}
```

BEISPIEL 13.79: Laden der Daten

C#
```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    InkCanvas1.Strokes.Clear();
    using (FileStream fs = new FileStream(@"c:\test.ink", FileMode.Open,
                                         FileAccess.Read))
    {
        InkCanvas1.Strokes = new StrokeCollection(fs);
        fs.Close();
    }
}
```

13.26.4 Konvertieren in eine Bitmap

Außer im *InkCanvas* können Sie mit den gesicherten Daten nichts anfangen. Da stellt sich schnell die Frage, wie Sie Ihre Kunstwerke in einem lesbaren Format sichern können. Am Beispiel des BMP-Formats wollen wir Ihnen die Vorgehensweise aufzeigen.

BEISPIEL 13.80: Sichern der aktuellen Zeichnung im BMP-Format

C#
```
private void Button_Click(object sender, RoutedEventArgs e)
{
    using (FileStream fs = new FileStream(@"c:\test.bmp",FileMode.Create,
                                          FileAccess.Write))
    {
```

Zielbitmap entsprechend der Größe der Zeichenfläche erzeugen:

```
        RenderTargetBitmap rtb = new RenderTargetBitmap((int)InkCanvas1.ActualWidth,
                                 (int)InkCanvas1.ActualHeight, 0, 0,
```

BEISPIEL 13.80: Sichern der aktuellen Zeichnung im BMP-Format

```
                                PixelFormats.Default);
```

Daten ausgeben:

```
                rtb.Render(InkCanvas1);
```

Mit dem BMP-Encoder kodieren:

```
                BmpBitmapEncoder encoder = new BmpBitmapEncoder();
                encoder.Frames.Add(BitmapFrame.Create(rtb));
```

Speichern:

```
                encoder.Save(fs);
                fs.Close();
            }
        }
```

13.26.5 Weitere Eigenschaften

Mit Hilfe der Eigenschaft *IsHighlighter* schalten Sie den Stift in einen teiltransparenten Modus, sodass darunter liegende Zeichnungen sichtbar bleiben.

Verwenden Sie Zeichentablets (z.B. von Watcom), hat der Anpressdruck des Stift einen Einfluss auf die Linienbreite. Setzen Sie *IgnorePressure* auf *True,* um dies zu verhindern.

Nicht in jedem Fall ist die aktuelle Zeichnung komplett zu sehen, bzw. Einzelheiten sind so klein, dass sie nicht sichtbar sind. Hier hilft die Verwendung einer Layout-Transformation weiter (Sie erinnern sich, dass es sich bei allen WPF-Controls um Vektorgrafiken handelt).

BEISPIEL 13.81: Skalieren des *InkCanvas* mit Hilfe eines *Slider*-Controls

XAML

```
    <InkCanvas Name="InkCanvas1">
      <InkCanvas.LayoutTransform>
        <ScaleTransform ScaleX="{Binding ElementName=slider2,Path=Value}"
                        ScaleY="{Binding ElementName=slider2,Path=Value}" />
      </InkCanvas.LayoutTransform>
    </InkCanvas>
...
    <Slider Height="26" Name="slider2" Maximum="5" Minimum=".1" Value="1"  Width="120" />
```

Ergebnis

Die Skalierung in Aktion:

13.27 Ellipse, Rectangle, Line und Co.

Fast hätten wir Sie vergessen, aber für die Oberflächengestaltung werden neben den komplexeren Controls auch die einfachen Elemente wie

- Ellipse/Kreis (*Ellipse*),
- Rechteck (*Rectangle*) und
- Linien (*Line*)

benötigt. Wie jedem anderen Control auch, können Sie diesen grafischen Primitiven auch Ereignisse zuordnen, um zum Beispiel auf einen Maus-Klick zu reagieren.

13.27.1 Ellipse

Für das Zeichnen von Kreisen oder Ellipsen nutzen Sie das *Ellipse*-Control. Die Eigenschaften *Height* und *Width* sind, sicher wenig überraschend, für die Abmessungen der Ellipse verantwortlich, die Linienfarbe stellen Sie über die Eigenschaft *Stroke,* die Linienstärke mit *StrokeThickness* ein. Die Füllung bestimmen Sie über *Fill*.

BEISPIEL 13.82: Verwendung *Ellipse*

XAML

```
<Ellipse Width="50" Height="80" Stroke="Blue" StrokeThickness="5" Fill="Aqua"/>
<Ellipse Width="80" Height="80" Stroke="Black" StrokeThickness="2" Fill="Yellow"
         MouseDown="Ellipse_MouseDown"/>
```

C#

Die Ereignismethode:

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e)
{
   MessageBox.Show("MouseDown");
}
```

Ergebnis

Die Ausgaben:

13.27.2 Rectangle

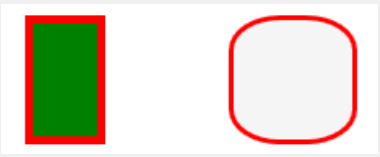
Auch hier bestimmen *Height* und *Width* die äußeren Abmessungen, *Stroke* und *StrokeThickness* legen Farbe und Linienstärke fest. Auch eine Eigenschaft *Fill* für die Füllung steht wieder zur Verfügung. Zusätzlich bieten die Eigenschaften *RadiusX* und *RadiusY* die Möglichkeit, einen Eckradius zu definieren.

BEISPIEL 13.83: Verwendung *Rectangle*

XAML

```
<Rectangle Width="50" Height="80" Stroke="Red" StrokeThickness="5" Fill="Green"/>
<Rectangle Width="80" Height="80" RadiusX="30" RadiusY="20" Stroke="Red"
           StrokeThickness="3" Fill="WhiteSmoke"/>
```

Ergebnis



13.27.3 Line

Auch das Zeichnen von Linien wollen wir nicht vergessen, mit *X1*,*Y1* und *X2*,*Y2* legen Sie Start- und Endpunkt der Linie fest. *Stroke* und *StrokeThickness* stellen auch hier Farbe und Linienstärke ein.

BEISPIEL 13.84: Verwendung *Line*

XAML

```
<Line X1="10" Y1="10" X2="100" Y2="100" Stroke="Black" StrokeThickness="3"/>
<Line X1="10" Y1="100" X2="100" Y2="10" Stroke="Red" StrokeThickness="5"/>
```

Ergebnis

Die beiden gezeichneten Linien:

HINWEIS: Mehr Informationen zur Grafikausgabe und zu den grafischen Primitiven finden Sie in unserem [Visual C# 2012 Kochbuch].

13.28 Browser

Wie kann es anders sein, auch in WPF findet der ambitionierte Programmierer ein entsprechendes Control wieder, das die Funktionalität des Microsoft Internet Explorers kapselt. Im Gegensatz zu seinem Windows Forms Pendant verfügt das still und heimlich mit der .NET-Version 3.3 SP1 eingeführte Control über einige recht interessante Methoden, die im Zusammenhang mit der Darstellung von Datenbankinhalten von Interesse sind.

Die Verwendung selbst ist recht einfach, es genügt, wenn Sie der Eigenschaft *Source* eine entsprechende Webadresse zuweisen, damit deren Inhalte dargestellt werden.

BEISPIEL 13.85: Adresse per XAML-Code zuweisen

XAML

```
<WebBrowser Name="webBrowser1" Source="http://www.spiegel.de" />
```

BEISPIEL 13.86: Einfaches Browserformular mit Eingabezeile und Adressübergabe per Code

XAML

```
<Window x:Class="BSP_Controls.Browser_Bsp"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Browser_Bsp" Height="362" Width="635">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <TextBox Name="txt1" FontSize="14" Margin="0,0,0,5"
                KeyDown="txt1_KeyDown">http://www.doko-buch.de</TextBox>
        </StackPanel>
        <WebBrowser Name="webBrowser1" />
    </DockPanel>
</Window>
```

C#

Die Adressübergabe zur Laufzeit:

```
private void txt1_KeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter)
        webBrowser1.Source = new Uri(txt1.Text);
}
```

Ergebnis

Alternativ können Sie die Navigation zur Zielseite auch mit den Methoden

- *Navigate* (Uri übergeben),
- *NavigateToStream* (lädt die Inhalte aus einem *Stream*) oder
- *NavigateToString* (lädt die Inhalte aus dem übergebenen *String*)

starten. Interessant ist hier vor allem *NavigateToStream,* was im Zusammenhang mit der Darstellung von Blob-Daten aus Ressourcen, Datenbanken oder Webdiensten recht interessant ist.

BEISPIEL 13.87: Laden von Ressourcedaten

C#

Laden einer eingebetteten HTML-Seite (*info.htm*):

```
using System.IO;
...
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
            Uri uri = new Uri(@"pack://application:,,,/info.htm", UriKind.Absolute);
            Stream src = Application.GetResourceStream(uri).Stream;
            webBrowser1.NavigateToStream(src);
        }
```

Wie Sie auch direkt Strings für das Erstellen der HTML-Seite verwenden können, zeigt folgendes einfaches Beispiel:

BEISPIEL 13.88: Verwendung von *NavigateToString*

C#

```
    public partial class Browser_Bsp : Window
    {
        public Browser_Bsp()
        {
            InitializeComponent();
            webBrowser1.NavigateToString("<HTML><BODY>Ein einfaches" +
             " <b>HTML-Dokument</b><br> mit wenig <u>Text</u>.</BODY></HTML>");
        }
```

Ergebnis

Auch die *Refresh*-Methode für das erneute Laden einer Seite sollten wir nicht vergessen.

Über den aktuellen Stand der Dinge können Sie sich mit Hilfe der drei Ereignisse *Navigating*, *Navigated* und *LoadCompleted* informieren.

Die Navigation zwischen mehreren Seiten ist über die Methoden *GoBack* und *GoForward* möglich. Ob der Aufruf der Methoden zulässig ist, bestimmen Sie mit den Eigenschaften *CanGoBack* und *CanGoForward*.

13.29 Ribbon

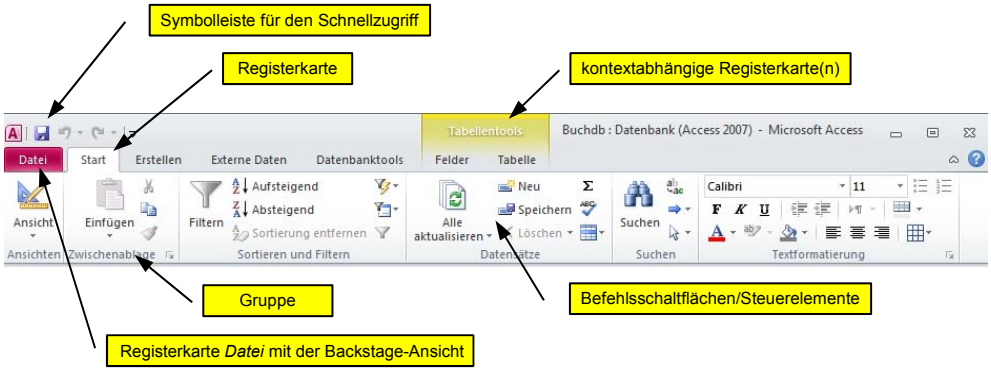
Bereits mit Office 2007 wurde von Microsoft das weit verbreitete Menü- und Toolbar-Konzept komplett über den Haufen geworfen, seitdem übernimmt der *Ribbon* (bzw. auf Neudeutsch das Menüband[1]) diese Funktion.

Aus guten Gründen scheiden sich an diesem Konzept die Geister – der angeblich intuitiven Funktionsweise steht auf der anderen Seite ein wesentlich erhöhter Platzbedarf und eine geringe Übersichtlichkeit mit langen Mauswegen gegenüber. Aber natürlich ist der Ribbon viel bunter und moderner, ein Trend der auch bei den neuen Windows 8-Oberflächen zu beobachten ist ...

Auch Ihre WPF-Anwendungen können von den Vorzügen des *Ribbon* profitieren, bietet doch Microsoft eine entsprechende Library.

13.29.1 Allgemeine Grundlagen

Bevor wir uns der eigentlichen Programmierung widmen, wollen wir uns zunächst einmal näher mit einem *Ribbon* und den entsprechenden Begrifflichkeiten beschäftigen, andernfalls kann es schnell zu Missverständnissen kommen. Die folgende Abbildung zeigt das "Opfer" unserer anschließenden Programmierversuche am Beispiel von Microsoft Access 2010:

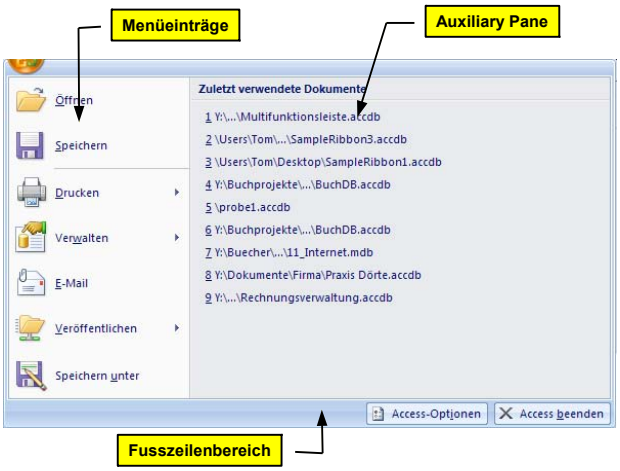


- Registerkarte *Datei*
  Diese ersetzt im Wesentlichen den früheren Office-Button bzw. das noch ältere *Datei*-Menü. Hinter der Registerkarte verbirgt sich in den neueren Office-Versionen die so genannte *Backstage*-Ansicht. **Diese können Sie mit dem vorliegenden Ribbon-Control nicht realisieren.** Sie haben nur die Möglichkeit, ein Anwendungsmenü zu erstellen, dieses entspricht der Ansicht in den Office 2007-Applikationen.
- Symbolleiste für den Schnellzugriff
  Hier können Sie kleine Schaltflächen einblenden, die dem Anwender immer zur Verfügung stehen sollen.

[1] Dieses wurde zwischenzeitlich auch als "Multifunktionsleiste" kurz "MuFu" bezeichnet..

- Registerkarten
  Diese stellen aufgabenorientierte Funktionen zur Verfügung.
- Gruppen
  Diese teilen die Funktionen innerhalb der Registerkarten in einzelne Bereiche.
- Befehlsschaltflächen und andere Steuerelemente

Das Anwendungsmenü selbst gliedert sich in eine Reihe von Menüeinträgen (*Menu Items*), den Zusatzbereich (Auxiliary Pane) sowie den Fußzeilenbereich (*Footer Pane*):



Damit schließen wir unsere kurzen theoretischen Erörterungen zum *Ribbon* ab und wenden uns der Praxis zu.

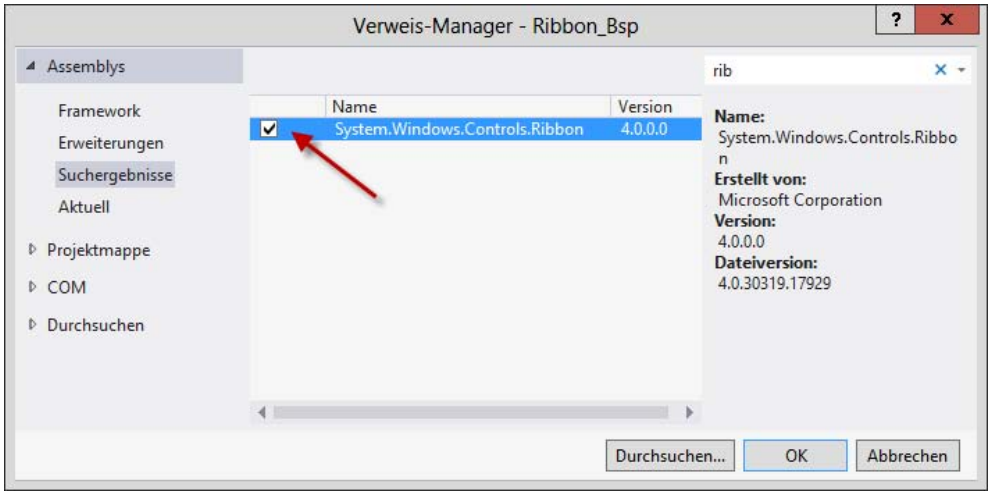
13.29.2 Download/Installation

Waren Sie in den bisherigen Versionen darauf angewiesen, sich die Ribbon-Library nachträglich herunterzuladen und zu installieren, ist diese seit Visual Studio 2012 bereits in der Standardinstallation enthalten.

Wer noch mit einer früheren Version von Visual Studio arbeitet, muss die Library nachinstallieren. Sie finden diese unter der folgenden Adresse:

LINK: `http://www microsoft.com/en-us/download/details.aspx?id=11877`

Für das erfolgreiche Einbinden des Ribbons in Ihr WPF-Projekt müssen Sie diesem noch einen Verweis auf *System.Windows.Controls.Ribbon* hinzufügen:



Damit sind die Vorbereitungen abgeschlossen, wir können uns der Umsetzung zuwenden.

13.29.3 Erste Schritte

Haben Sie den Verweis auf die Assembly erfolgreich eingebunden, steht den ersten Gehversuchen nichts mehr im Wege. Öffnen Sie ein *Window* und positionieren Sie ein *Ribbon*-Element am oberen Fensterrand. Sie können dazu ein *DockPanel* oder ein *Grid* nutzen. Wir haben uns für die zweite Variante entschieden:

BEISPIEL 13.89: Einfacher *Ribbon* im Standard-Window

XAML

```
<Window x:Class="Ribbon_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

Gliedern der Seite:

```
    <Grid>
        <Grid.RowDefinitions>
```

Der *Ribbon* bestimmt die Höhe der ersten Zeile im Grid:

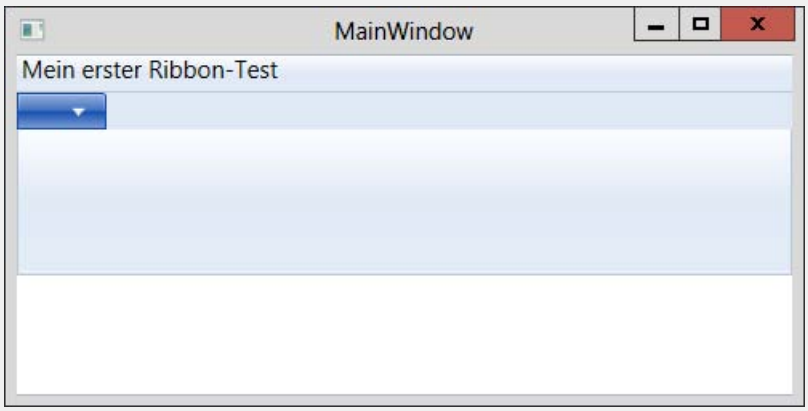
```
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
```

Den *Ribbon* der ersten Zeile zuordnen:

```
        <Ribbon Grid.Row="0" Title="Mein erster Ribbon-Test">
        </Ribbon>
    </Grid>
</Window>
```

BEISPIEL 13.89: Einfacher *Ribbon* im Standard-Window

Ergebnis



Wie Sie sehen, sehen Sie zunächst nicht viel. Lediglich ein Titel, die Schaltfläche für das Anwendungsmenü (funktionslos) und die Platzhalter für die späteren Registerkarten werden dargestellt.

Auf den Titel sollten Sie im Allgemeinen verzichten, dieser kostet nur Platz und hat keinen echten Nutzen (der Window-Titel sollte eigentlich reichen).

13.29.4 Registerkarten und Gruppen

Nächster Schritt ist das Erzeugen eigener Registerkarten, in die wiederum die Gruppen eingefügt werden. Die eigentlichen Befehlselemente sind dann den Gruppen zuzuordnen.

BEISPIEL 13.90: Registerkarten und Gruppen erzeugen

XAML

```
...
        <Ribbon Grid.Row="0" >
```

Erst die Registerkarte:

```
            <RibbonTab Header="Registerkarte 1" >
```

Dann die Gruppen:

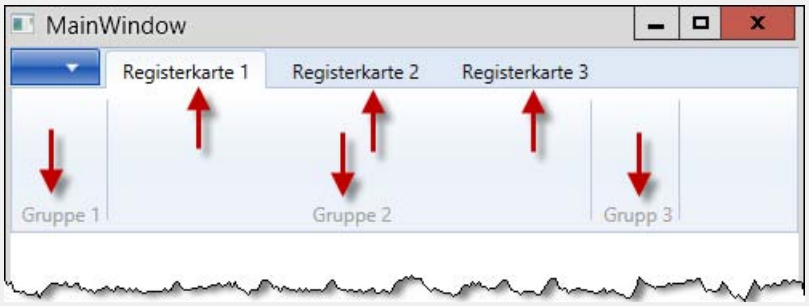
```
                <RibbonGroup Header="Gruppe 1">
                </RibbonGroup>
                <RibbonGroup Header="Gruppe 2" Width="300">
                </RibbonGroup>
                <RibbonGroup Header="Grupp 3">
                </RibbonGroup>
            </RibbonTab>
            <RibbonTab Header="Registerkarte 2">
            </RibbonTab>
            <RibbonTab Header="Registerkarte 3">
            </RibbonTab>
```

BEISPIEL 13.90: Registerkarten und Gruppen erzeugen

Die Sichtbarkeit von Gruppen oder ganzen Registerkarten können Sie mit der Eigenschaft *Visibility* steuern, ein Ein-/Ausblenden zur Laufzeit ist jederzeit möglich:

```
            <RibbonTab Header="Registerkarte 4" Visibility="Collapsed">
            </RibbonTab>
        </Ribbon>
...
```

Ergebnis



13.29.5 Kontextabhängige Registerkarten

Neben den meist standardmäßig angezeigten Registerkarten gibt es auch kontextabhängige Registerkarten (meist farblich markiert), die im Zusammenhang mit Auswahlvorgängen oder Selektionen des Nutzers zusätzlich eingeblendet werden. Auch diese können Sie selbst realisieren:

BEISPIEL 13.91: Kontextabhängige Registerkarten erzeugen

XAML

```
...
        <Ribbon Grid.Row="0" >
```

Hier definieren wir die Gruppen mit ihren Farben:

```
            <Ribbon.ContextualTabGroups>
                <RibbonContextualTabGroup Header="Kontextgruppe 1"
                                          Visibility="Visible" Background="#FFFFE90C" >
                </RibbonContextualTabGroup>
            </Ribbon.ContextualTabGroups>
...
```

Später können wir auf die obigen Gruppen Bezug nehmen:

```
            <RibbonTab Header="Registerkarte 5" ContextualTabGroupHeader="Kontextgruppe 1">
            </RibbonTab>
            <RibbonTab Header="Registerkarte 6" ContextualTabGroupHeader="Kontextgruppe 1">
            </RibbonTab>
...
```

BEISPIEL 13.91: Kontextabhängige Registerkarten erzeugen

Ergebnis

13.29.6 Einfache Beschriftungen

Möchten Sie innerhalb der Gruppen kurze Texte anzeigen, können Sie dafür das *RibbonTwoLineText*-Element nutzen:

BEISPIEL 13.92: Beschriftungen realisieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <RibbonTab Header="Registerkarte 1" >
...
                <RibbonGroup Header="Gruppe 2" Width="300">
                    <RibbonTwoLineText Text="Textzeile 1"/>
                    <RibbonTwoLineText Text="Textzeile 2"/>
                </RibbonGroup>
...
```

Ergebnis

13.29.7 Schaltflächen

Die Ribbon-Library bietet vier verschiedene Varianten von Schaltflächen an:

- nur kleines Bild (16x16 Pixel)
- kleines Bild und Text
- nur großes Bild (32x32 Pixel)
- großes Bild und Text

Steuern können Sie das Aussehen über die Attribute *Label* (die Beschriftung), *SmallImageSource* und/oder *LargeImageSource*.

BEISPIEL 13.93: Schaltflächen definieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.ContextualTabGroups>
...
            <RibbonTab Header="Registerkarte 1" >
                <RibbonGroup Header="Gruppe 1">
                    <RibbonButton Label="Beschriftung 1"
                                  LargeImageSource="Images/printer.png" />
                    <RibbonButton Label="Beschriftung 2"
                                  SmallImageSource="Images/page_paste.png" />
                    <RibbonButton Label="Beschriftung 3" />
                    <RibbonButton SmallImageSource="Images/arrow_down.png"/>
                </RibbonGroup>
                <RibbonGroup Header="Gruppe 2" Width="300">
...
```

Ergebnis

Als Alternative für den einfachen *RibbonButton* bietet sich noch der *RibbonToggleButton* an. Dieser hat prinzipiell das gleiche Aussehen und dieselben Konfigurationsmöglichkeiten, kann aber zwei Schaltzustände darstellen:

BEISPIEL 13.94: Verwendung *RibbonToggleButton*

C#

```
...
                <RibbonGroup Header="Gruppe 2" Width="300">
                    <RibbonTwoLineText Text="Textzeile 1"/>
                    <RibbonTwoLineText Text="Textzeile 2"/>
                    <RibbonTwoLineText Text="Textzeile 3"/>
                    <RibbonToggleButton Label="Beschriftung 1"
                          SmallImageSource="Images/lock.png"   IsChecked="True"  />
                    <RibbonToggleButton Label="Beschriftung 2"
```

BEISPIEL 13.94: Verwendung *RibbonToggleButton*

```
                    SmallImageSource="Images/lock.png"  />
                </RibbonGroup>
...
```

Ergebnis

Schaltflächen zusammenfassen

Möchten Sie mehrere Schaltflächen so anordnen, dass ein Zusammenhang zwischen diesen hergestellt wird, können Sie die *RibbonControlGroup* verwenden. Prominentes Beispiel dürfte die Zuordnung der Textausrichtung sein:

BEISPIEL 13.95: Verwendung *RibbonControlGroup*

C#

```
...
                <RibbonControlGroup>
                    <RibbonButton SmallImageSource="Images/text_align_center.png"  />
                    <RibbonButton SmallImageSource="Images/text_align_justify.png"  />
                    <RibbonButton SmallImageSource="Images/text_align_left.png"  />
                    <RibbonButton SmallImageSource="Images/text_align_right.png"  />
                </RibbonControlGroup>
...
```

Ergebnis

13.29.8 Auswahllisten

Hier bietet sich die *RibbonComboBox* an. Diese wird im *Ribbon* mit einer *RibbonGallery* kombiniert. Dies hat den Vorteil, dass die Einträge auch mehrspaltig und gruppiert dargestellt werden können.

BEISPIEL 13.96: Darstellen von Listen im Ribbon

C#

...

Wir kombinieren *RibbonComboBox* und *RibbonGallery*:

```
                <RibbonComboBox SelectionBoxWidth="100" IsEditable="True"
                     Name="RibbonComboBox1">
                    <RibbonGallery SelectedValue="Eintrag 4" SelectedValuePath="Content">
```

BEISPIEL 13.96: Darstellen von Listen im Ribbon

Eine erste Rubrik erzeugen:

```
                        <RibbonGalleryCategory Header="Wählen Sie einen Eintrag"
                                    MaxColumnCount="2">
                            <RibbonGalleryItem Content="Eintrag 1" Foreground="Green" />
                            <RibbonGalleryItem Content="Eintrag 2" Foreground="Red" />
                            <RibbonGalleryItem Content="Eintrag 3" Foreground="Blue" />
                            <RibbonGalleryItem Content="Eintrag 4" Foreground="Black" />
                            <RibbonGalleryItem Content="Eintrag 5" Foreground="Black" />
                            <RibbonGalleryItem Content="Eintrag 6" Foreground="Black" />
                            <RibbonGalleryItem Content="Eintrag 7" Foreground="Black" />
                        </RibbonGalleryCategory>
```

Eine zweite Rubrik erzeugen:

```
                        <RibbonGalleryCategory  Header="Rubrik 2"  MaxColumnCount="2">
                            <RibbonGalleryItem Content="Eintrag 1"/>
                            <RibbonGalleryItem Content="Eintrag 2" />
                            <RibbonGalleryItem Content="Eintrag 3" />
                            <RibbonGalleryItem Content="Eintrag 4" />
                            <RibbonGalleryItem Content="Eintrag 5" />
                            <RibbonGalleryItem Content="Eintrag 6" />
                            <RibbonGalleryItem Content="Eintrag 7" />
                        </RibbonGalleryCategory>
                    </RibbonGallery>
                </RibbonComboBox>
```

Diese Liste füllen wir zur Laufzeit:

```
                <RibbonComboBox SelectionBoxWidth="100" IsEditable="False"
                                Name="RibbonComboBox2" >
```

Beachten Sie, dass wir auf Änderungen in der *RibbonGallery*, nicht in der *RibbonComboBox* reagieren müssen:

```
                    <RibbonGallery SelectedValue="Zeile 4"
                                   SelectedValuePath="Content"
                                   SelectionChanged="RibbonGallery_SelectionChanged_1" >
                        <RibbonGalleryCategory  Header="Kategorie 1"
                             MaxColumnCount="1" Name="RibbonGalleryCategory1"/>
                        <RibbonGalleryCategory  Header="Kategorie 2"
                             MaxColumnCount="1" Name="RibbonGalleryCategory2"/>
                    </RibbonGallery>
                </RibbonComboBox>
...
```

C#

```
...
 public partial class MainWindow : RibbonWindow
    {
        public MainWindow()
```

BEISPIEL 13.96: Darstellen von Listen im Ribbon

```
        {
            InitializeComponent();
```

Die *RibbonGallery* in zwei Kategorien füllen:

```
            for (int i = 0; i < 10; i++)
                RibbonGalleryCategory1.Items.Add("Zeile " + i.ToString());
            for (int i = 0; i < 6; i++)
                RibbonGalleryCategory2.Items.Add("Zeile " + i.ToString());
        }
```

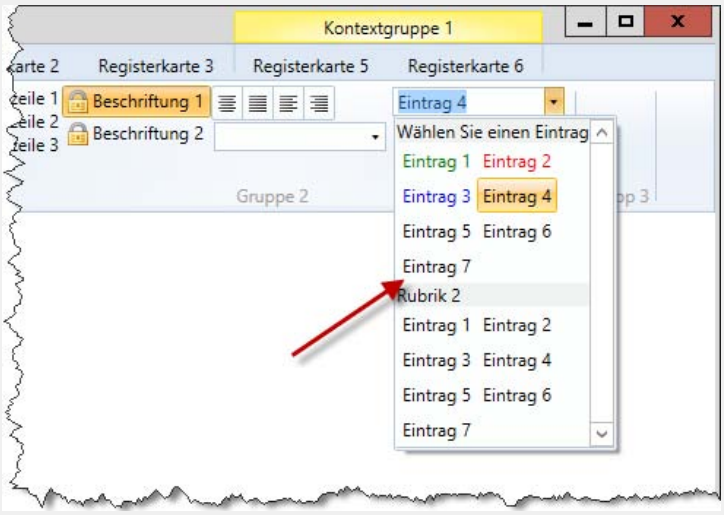
Anzeige der neuen Auswahl:

```
        private void RibbonGallery_SelectionChanged_1(object sender,
                                RoutedPropertyChangedEventArgs<object> e)
        {
```

Bestimmen der Auswahl:

```
            MessageBox.Show(e.NewValue.ToString());
        }
    }
...
```

Ergebnis



Eine *RibbonGallery* können Sie auch in den folgenden Steuerelementen definieren:

- *RibbonSplitMenuButton*
- *RibbonApplicationSplitMenuItem*
- *RibbonSplitMenuItem*

Wir gehen nicht im Einzelnen darauf ein, dies würde den Rahmen diese Kapitels sprengen.

13.29.9 Optionsauswahl

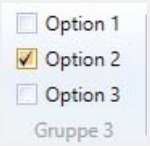
Als Alternative zum *RibbonToggleButton* bietet sich die *RibbonCheckBox* an. Diese wird wie ihr WPF-Pendant konfiguriert und ausgewertet:

BEISPIEL 13.97: Verwendung *RibbonCheckBox*

XAML

```
...
                <RibbonGroup Header="Gruppe 3">
                    <RibbonCheckBox Label="Option 1"/>
                    <RibbonCheckBox Label="Option 2" IsChecked="True"/>
                    <RibbonCheckBox Label="Option 3"/>
                </RibbonGroup>
...
```

Ergebnis



13.29.10 Texteingaben

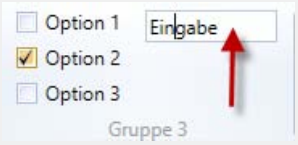
Hinter der *RibbonTextBox* verbirgt sich nichts anderes als das allseits bekannte Textfeld, das bekanntlich der Eingabe von Zeichenketten dient. Allerdings sollten Sie hier nicht allzu viel erwarten, die diversen Beschränkungen und Ereignisse wie bei normalen Textfeldern sind nicht realisierbar. Aber das ist sicher auch nicht der eigentliche Zweck dieses Controls.

BEISPIEL 13.98: Verwendung *RibbonTextBox*

XAML

```
...
                <RibbonGroup Header="Gruppe 3">
                    <RibbonCheckBox Label="Option 1"/>
                    <RibbonCheckBox Label="Option 2" IsChecked="True"/>
                    <RibbonCheckBox Label="Option 3"/>
                    <RibbonTextBox Text="Eingabe" MaxLength="10"  />
                </RibbonGroup>
...
```

Ergebnis



13.29.11 Screentips

Auch wenn Sie es mitunter lästig finden, für den unerfahrenen Endanwender sind die kleinen Hilfetexte, die in den Screentips angezeigt werden, meist unentbehrlich. Steuern können Sie die

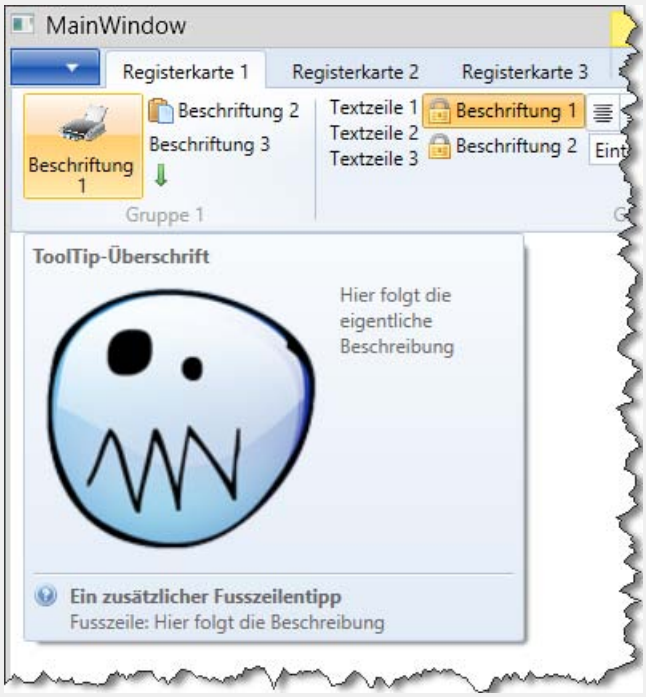
Inhalte über die Attribute *ToolTipTitle*, *ToolTipImageSource*, *ToolTipDescription*, *ToolTipFooterTitle*, *ToolTipFooterImageSource* und *ToolTipFooterDescription*.

BEISPIEL 13.99: Einen Screentip realisieren

XAML

```
...
    <RibbonButton Label="Beschriftung 1"
                  LargeImageSource="Images/printer.png"
                  ToolTipTitle="ToolTip-Überschrift"
                  ToolTipImageSource="Images/user.png"
                  ToolTipDescription="Hier folgt die eigentliche Beschreibung"
                  ToolTipFooterTitle="Ein zusätzlicher Fusszeilentipp"
                  ToolTipFooterImageSource="Images/help.png"
                  ToolTipFooterDescription="Fusszeile: Hier folgt die Beschreibung" />
...
```

Ergebnis



13.29.12 Symbolleiste für den Schnellzugriff

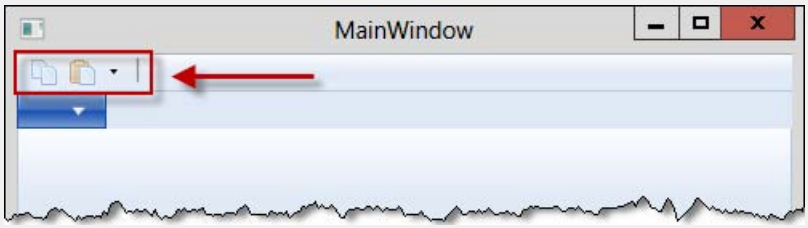
Die Symbolleiste für den Schnellzugriff stellt dem Anwender jederzeit einige wichtige Funktionen im oberen Fensterbereich zur Verfügung. Fügen Sie in den *Ribbon* ein *RibbonQuickAccessToolBar*-Element ein, dieses enthält dann die *RibbonButton*- oder *RibbonSplitButton*-Elemente.

BEISPIEL 13.100: Symbolleiste für den Schnellzugriff implementieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.QuickAccessToolBar>
                <RibbonQuickAccessToolBar>
                    <RibbonButton SmallImageSource="Images/page_copy.png"/>
                    <RibbonSplitButton SmallImageSource="Images/page_paste.png"/>
                </RibbonQuickAccessToolBar>
            </Ribbon.QuickAccessToolBar>
        </Ribbon>
...
```

Ergebnis



Achten Sie darauf, dass Sie den beiden Schaltflächen die Eigenschaft *SmallImageSource* zuweisen, nicht *LargeImageSource*, es sollen ja auch nur kleine Schaltflächen angezeigt werden. Auf die Angabe einer Beschriftung können Sie aus naheliegenden Gründen ebenfalls verzichten.

13.29.13 Das RibbonWindow

Leider entspricht das Ergebnis unserer Bemühungen nicht ganz den Erwartungen, die Schnellzugriffsleiste wird nicht, wie erwartet, in der Titelzeile des Fensters eingeblendet, sondern darunter.

Besser klappt es mit der *RibbonWindow*-Klasse, die diese Aufgabe übernimmt. *RibbonWindow* integriert die Schnellzugriffsleiste wie gewünscht in die Kopfzeile, die nötigen Anpassungen Ihres Programms zeigt das folgende Beispiel.

BEISPIEL 13.101: Verwenden der *RibbonWindow*-Klasse

XAML

```
<RibbonWindow x:Class="Ribbon_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="250" Width="500" WindowStartupLocation="CenterScreen">
...
</RibbonWindow>
```

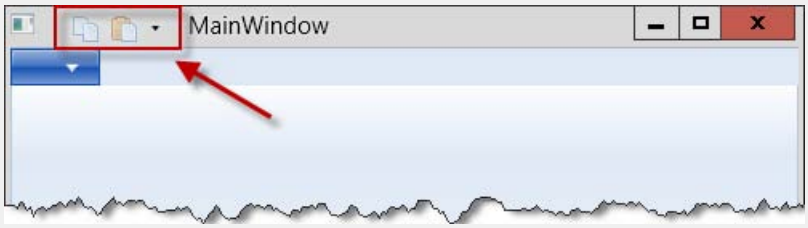
C#

```
...
using System.Windows.Controls.Ribbon;
...
    public partial class MainWindow : RibbonWindow
    {
...
```

BEISPIEL 13.101: Verwenden der *RibbonWindow*-Klasse

Ergebnis

Das hier sieht schon eher nach dem bekannten Ribbon aus:



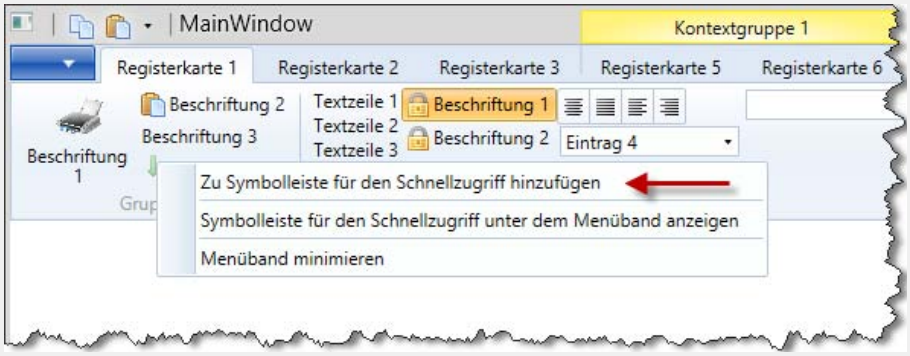
Ist einer Schaltfläche im normalen Ribbon ein *Command* zugeordnet und setzen Sie die Eigenschaft *CanAddToQuickAccessToolBarDirectly* dieser Schaltfläche auf *True*, kann der Anwender diese Schaltfläche zur Laufzeit der Schnellzugriffsleiste hinzufügen (Kontextmenü, rechte Maustaste).

BEISPIEL 13.102: Schnellzugriffsleiste zur Laufzeit erweitern

XAML

```
...
 <RibbonButton SmallImageSource="Images/arrow_down.png"
               CanAddToQuickAccessToolBarDirectly="True"
               Command="ApplicationCommands.Close"/>
...
```

Ergebnis



13.29.14 Menüs

Ganz nebenbei hat der *Ribbon* auch einen *RibbonMenuButton* zu bieten, was allerdings gleich die Frage nach dem Sinn auslöst, soll das Menüband doch die guten alten Menüs ersetzen.

Sie können dieses Steuerelement als Container für eingelagerte *RibbonMenuItem*-Elemente nutzen oder aber auch die schon bekannten Ribbon-Controls darin unterbringen. Zusätzlich bietet ein *RibbonSeparator* die Möglichkeit, die einzelnen Controls optisch voneinander zu trennen. Dies kann als simpler Strich oder auch als Beschriftung erfolgen, wie es das folgende Beispiel zeigt.

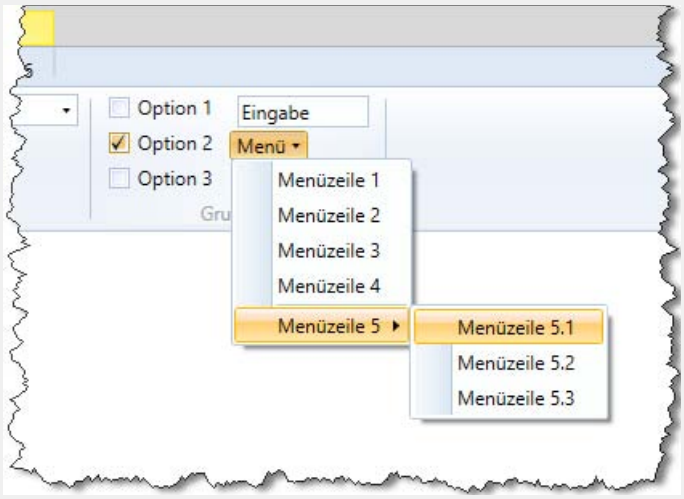
Genügt Ihnen eine Menüebene nicht, lassen sich weitere *RibbonMenuItem*-Steuerelemente hierarchisch eingliedern, Sie erhalten dann die bekannten Untermenüs.

BEISPIEL 13.103: Ein Menü implementieren

XAML

```
...
                <RibbonMenuButton Label="Menü">
                    <RibbonMenuButton.Items>
                        <RibbonMenuItem Header="Menüzeile 1" />
                        <RibbonMenuItem Header="Menüzeile 2" />
                        <RibbonButton Label="Ein Button"/>
                        <RibbonMenuItem Header="Menüzeile 3" />
                        <RibbonMenuItem Header="Menüzeile 4" />
                        <RibbonSeparator/>
                        <RibbonMenuItem Header="Menüzeile 5" >
                            <RibbonMenuItem Header="Menüzeile 5.1" />
                            <RibbonMenuItem Header="Menüzeile 5.2" />
                            <RibbonMenuItem Header="Menüzeile 5.3" />
                        </RibbonMenuItem>
                    </RibbonMenuButton.Items>
                </RibbonMenuButton>
...
```

Ergebnis



HINWEIS: Erliegen Sie bitte nicht der Versuchung, ein mehrfach geschachteltes Menü zu erzeugen, dies ist der Übersicht sicher nicht zuträglich und widerspricht dem Grundkonzept des Menübands!

Nach diesem "Rundflug" über die verfügbaren Steuerelemente wollen wir uns noch mit einem Menü-Bereich beschäftigen, den wir bisher in unseren Betrachtungen sträflich vernachlässigt haben.

13.29.15 Anwendungsmenü

Leider steht in der vorliegenden Form des Ribbons nicht die vom aktuellen Microsoft Office bekannte Backstage-Ansicht zur Verfügung, Sie müssen mit dem einfacheren Anwendungsmenü Vorlieb nehmen. Dieses gliedert sich in eine Reihe von Schaltflächen auf der linken Seite, eine Detailbereich auf der rechten Seite und einem Fusszeilenbereich.

BEISPIEL 13.104: Ein Anwendungsmenü implementieren

XAML

```
...
        <Ribbon Grid.Row="0" >
            <Ribbon.QuickAccessToolBar>
            </Ribbon.QuickAccessToolBar>
            <Ribbon.ApplicationMenu>
```

Das Symbol für das Anwendungsmenü festlegen:

```
                <RibbonApplicationMenu SmallImageSource="Images/help.png">
```

Der erste Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP45.ico"
                        Header="Eintrag 1">
```

Dieser Eintrag hat weitere Untereinträge:

```
                        <RibbonApplicationMenuItem Header="Untereintrag 1.1"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.2"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.3"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.4"  />
                        <RibbonApplicationMenuItem Header="Untereintrag 1.5"  />
                    </RibbonApplicationMenuItem>
```

Zweiter Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP49.ico"
                                               Header="Eintrag 2"/>
```

Dritter Haupteintrag:

```
                    <RibbonApplicationMenuItem ImageSource="Images/CP09.ico"
                                               Header="Eintrag 3"/>
```

In den Detailbereich blenden wir eine Liste der zuletzt geöffneten Dateien ein:

```
                    <RibbonApplicationMenu.AuxiliaryPaneContent>
```

Ein *Grid*, um Überschrift und Liste zu trennen:

```
                        <Grid ScrollViewer.VerticalScrollBarVisibility="Auto">
```

Verzichten Sie nicht auf die automatische Anzeige der Scrollbars, ist die Liste länger, würden sonst Einträge abgeschnitten werden.

```
                            <Grid.RowDefinitions>
                                <RowDefinition Height="Auto"/>
```

BEISPIEL 13.104: Ein Anwendungsmenü implementieren

```
                        <RowDefinition Height="*"/>
                    </Grid.RowDefinitions>
```

Die kleine Kopfzeile:

```
                    <Border Grid.Row="0" BorderBrush="DarkBlue"
                            BorderThickness="0,0,0,1">
                        <Label Content="Zuletzt geöffnete Dateien"  />
                    </Border>
```

Die Liste befüllen wir zur Laufzeit:

```
                    <ListBox Grid.Row="1" Name="DateiListe1" />
                </Grid>
            </RibbonApplicationMenu.AuxiliaryPaneContent>
```

In der Fußzeile des Anwendungsmenüs blenden wir einen Copyright-Vermerk und eine Schaltfläche zum Beenden ein:

```
            <RibbonApplicationMenu.FooterPaneContent>
                <Grid Margin="2" >
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*"/>
                        <ColumnDefinition Width="Auto"/>
                    </Grid.ColumnDefinitions>
```

Der Text:

```
                    <TextBlock Grid.Column="0" VerticalAlignment="Stretch">
                             (c) DOKO-Buch</TextBlock>
```

Die Schaltfläche:

```
                    <RibbonButton Grid.Column="1" Label="Beenden"
                             SmallImageSource="Images/arrow_down.png"/>
                </Grid>
            </RibbonApplicationMenu.FooterPaneContent>
        </RibbonApplicationMenu>
    </Ribbon.ApplicationMenu>
...
```

C# Die Liste zur Laufzeit füllen:

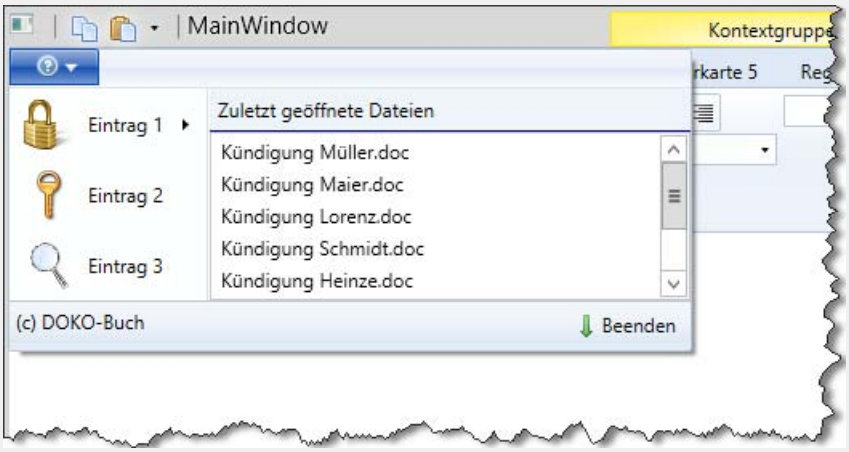
```
...
  public MainWindow()
      {
          InitializeComponent();
          DateiListe1.Items.Add("Kündigung Müller.doc");
          DateiListe1.Items.Add("Kündigung Maier.doc");
          DateiListe1.Items.Add("Kündigung Lorenz.doc");
          DateiListe1.Items.Add("Kündigung Schmidt.doc");
          DateiListe1.Items.Add("Kündigung Heinze.doc");
```

BEISPIEL 13.104: Ein Anwendungsmenü implementieren

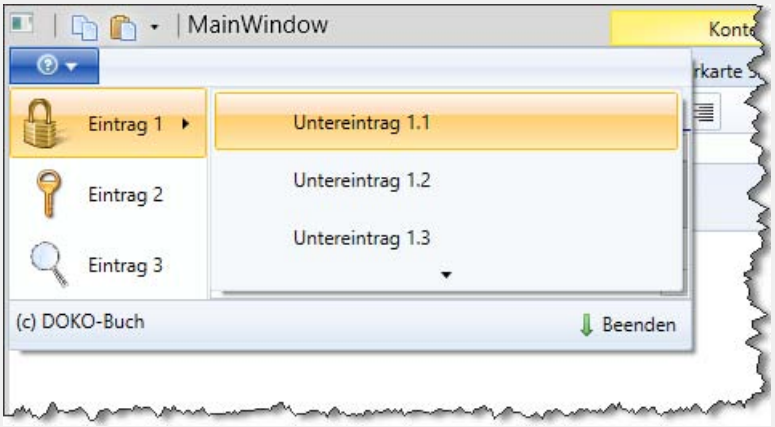
```
            DateiListe1.Items.Add("Kündigung Koch.doc");
            DateiListe1.Items.Add("Kündigung Walter.doc");
            DateiListe1.Items.Add("Kündigung Gewinnus.doc");
        }
...
```

Ergebnis

Das geöffnete Anwendungsmenü:



Die Untereinträge des ersten Menüpunktes:



Gerade beim Aufklappen der Menüeinträge wird schnell klar, dass hier die Übersichtlichkeit massiv leidet, da die Liste der letzten Dateien verdeckt wird. Verzichten Sie also besser auf derartige Verschachtelungen, zumal die Höhe des Untermenüs durch das Anwendungsmenü beschränkt ist.

13.29.16 Alternativen

Sie haben es vielleicht schon bemerkt – auch bei der aktuellen Ribbon-Library ist bei Microsoft der erste Elan frühzeitig erloschen, und so fehlen einige Funktionen, andere sind fehlerhaft etc. Eine relevante Pflege scheint auch nicht mehr stattzufinden.

Eine sinnvolle (kostenlose) Alternative finden Sie unter der folgenden Adresse:

LINK: `http://fluent.codeplex.com/`

13.30 Chart

Auch hier scheinen die Entwickler unter akutem Zeitmangel zu leiden, denn in Visual Studio 2012 ist die Komponente immer noch nicht enthalten, stattdessen finden Sie sie unter der Adresse

LINK: `http://wpf.codeplex.com/releases/view/40535`

bzw. per NuGet unter "WPF Toolkit".

Nach der Installation binden Sie einen Verweis auf *System.Windows.Controls.DataVisualization.Toolkit* in Ihr Projekt ein (*Projekt|Verweis hinzufügen*).

Ein kleines Beispiel zeigt die Möglichkeiten:

BEISPIEL 13.105: Kuchendiagramm erzeugen

XAML

```
...
<Window x:Class="WpfApplication2.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Namespace einbinden:

```
        xmlns:tk="clr-namespace:System.Windows.Controls.DataVisualization.Charting;
                 assembly=System.Windows.Controls.DataVisualization.Toolkit"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

Chart erzeugen:

```
        <tk:Chart Name="Chart1" Title="Kuchendiagramm">
            <tk:PieSeries DependentValuePath="Value" IndependentValuePath="Key"
                          ItemsSource="{Binding}" IsSelectionEnabled="True" />
        </tk:Chart>
    </Grid>
</Window>
...
```

C#

```
...
  public MainWindow()
        {
            InitializeComponent();
```

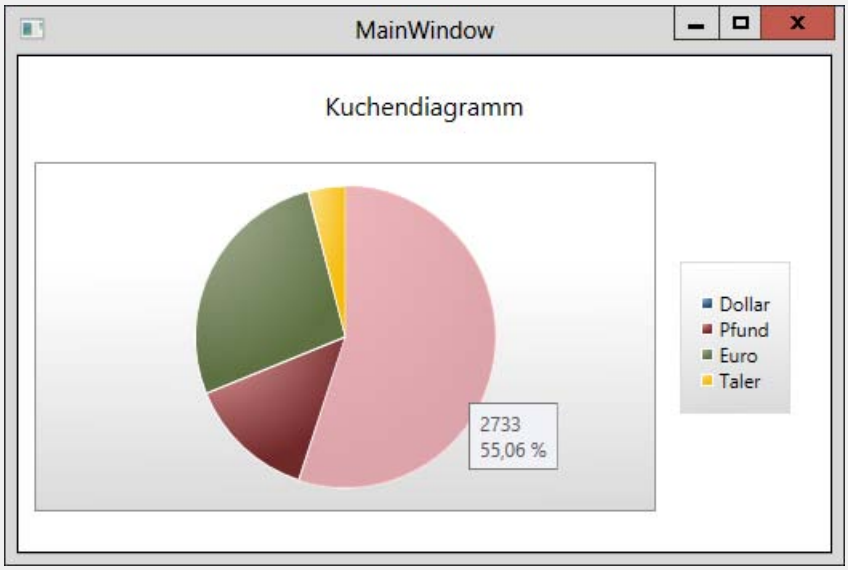
BEISPIEL 13.105: Kuchendiagramm erzeugen

```
            Dictionary<string, int> liste = new Dictionary<string, int>();
            liste.Add("Dollar", 2733);
            liste.Add("Pfund", 687);
            liste.Add("Euro", 1344);
            liste.Add("Taler", 200);
```

Daten zuweisen:

```
            this.DataContext = liste;
        }
...
```

Ergebnis

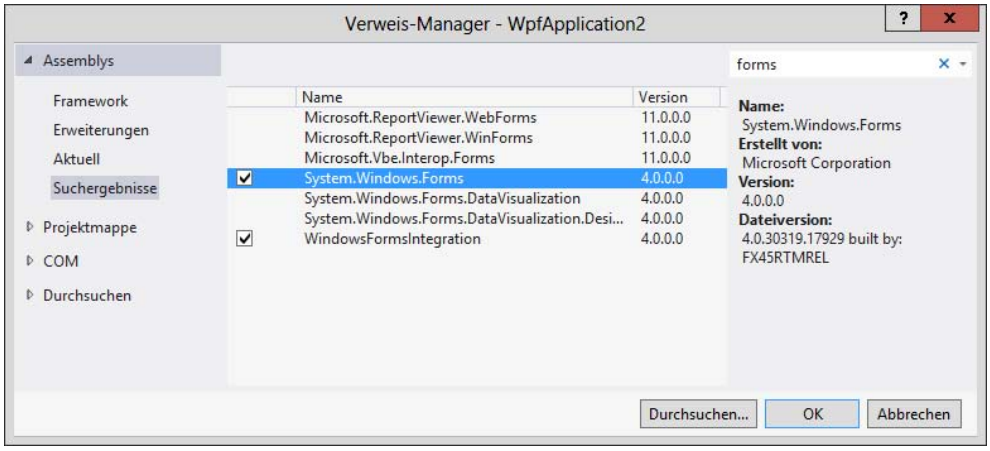


Bitte haben Sie Verständnis dafür, dass wir an dieser Stelle nicht alle Charttypen und Eigenschaften in epischer Breite vorstellen können. Im Internet finden Sie einige, mehr oder weniger aussagekräftige, Beispiele für die Arbeit mit dieser Library.

13.31 WindowsFormsHost

Wer immer noch seinen geliebten Windows Forms Controls nachtrauert, bekommt mit dem *WindowsFormsHost* die Möglichkeit, diese in eine WPF-Anwendung zu integrieren. Die Vorgehensweise entspricht etwa der beim entsprechenden Windows Forms Pendant.

Fügen Sie Ihrem Projekt zunächst einen Verweis auf die Assembly *System.Windows.Forms.dll* hinzu. Fügen Sie diesen Namespace sowohl der XAML als auch der Klassendefinition des gewünschten Fensters hinzu (siehe folgende Abbildung)



und setzen Sie das gewünschte Windows Forms-Control in ein *WindowsFormsHost*-Control (quasi als Container). Vergeben Sie einen Namen für das Windows Forms-Control, um im Quellcode auf dieses zugreifen zu können.

BEISPIEL 13.106: Die Windows Forms *CheckedListbox* in WPF nutzen

XAML

```
<Window x:Class="WindowsFormsHost_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Namespace einfügen für die Verwendung der Windows Forms-Controls:

```
        xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
        Title="Verwendung Windows Forms Host" Height="207" Width="266">
    <Grid>
```

Hier kommt der Host:

```
        <WindowsFormsHost Name="windowsFormsHost1">
```

Und hier können Sie das Windows Forms-Control einfügen (vergessen Sie nicht den Verweis auf den WindowsForms-Namespace):

```
            <wf:CheckedListBox x:Name="checkedListbox1"
                Width="200" Height="150" BackColor="Red" />
```

Beachten Sie die Namensvergabe über das *x:Name*-Attribut, andernfalls wird Ihr Control später nicht gefunden!

```
        </WindowsFormsHost>
    </Grid>
</Window>
```

BEISPIEL 13.106: Die Windows Forms *CheckedListbox* in WPF nutzen

C#

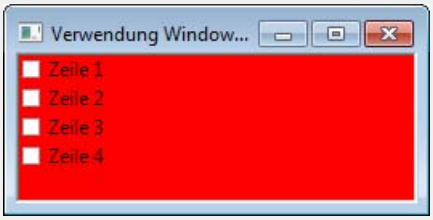
Namespace-Einbindung nicht vergessen:

```
...
using System.Windows.Forms;

namespace WindowsFormsHost_Bsp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
```

Und hier nutzen wir das Control wie gewohnt:

```
            InitializeComponent();
            checkedListbox1.Items.Add("Zeile 1");
            checkedListbox1.Items.Add("Zeile 2");
            checkedListbox1.Items.Add("Zeile 3");
            checkedListbox1.Items.Add("Zeile 4");
        }
    }
}
```

Ergebnis



Kapitel 14

Wichtige WPF-Techniken

Nachdem wir im vorhergehenden Kapitel einige ganz praktische Erfahrungen mit den WPF-Controls gesammelt haben, wollen wir uns jetzt mit den Besonderheiten der WPF-Programmierung im Vergleich zur Windows Forms-/Win32-Programmierung beschäftigen.

Damit Sie die WPF-Philosophie verstehen, wollen wir Ihnen zu Beginn einige wichtige Eckpfeiler dieser relativ neuen Technologie erklären.

14.1 Eigenschaften

Da die Definition und Verwendung von Eigenschaften in WPF etwas anders organisiert ist als in den bekannten Windows Forms-Anwendungen, sollten wir zunächst auf dieses Thema näher eingehen.

14.1.1 Abhängige Eigenschaften (Dependency Properties)

Mit den abhängigen (Dependency) und angehängten (Attached) Eigenschaften erweitert WPF das Spektrum der CLR-Eigenschaften. Abhängige Eigenschaften bieten gegenüber den "normalen" Eigenschaften folgende erweiterte Möglichkeiten:

- eine interne Prüfung (Validierung),
- automatisches Aktualisieren von Werten,
- die Verwendung von Callback-Methoden zur Signalisierung von Wertänderungen
- sowie die Vorgabe von Defaultwerten.

Notwendig wurde diese Erweiterung des Eigenschaftensystems, um viele der WPF-Features (Animationen, Datenbindung Styles etc.) zu realisieren. So werden beispielsweise Werte von Eigenschaften überwacht, Änderungen führen automatisch zu Änderungen an den abhängigen Objekten.

Abhängige Eigenschaften werden nicht als private Felder definiert, sondern als statische, öffentliche Instanzen der Klasse *System.Windows.DependencyProperty*, die über *get*- und *set*-Methoden angesprochen werden. Die Verwaltung der Eigenschaft wird vom WPF-Subsystem übernommen

(daher auch die *Register*-Methode, siehe folgendes Beispiel), die der Eigenschaft übergeordnete Klasse stellt quasi nur noch eine Schnittstelle zu dieser Eigenschaft zur Verfügung.

Neben dem Wert können mit dem Default-Wert und der Callback-Methode auch weitere Metadaten zu einer Eigenschaft gespeichert werden.

BEISPIEL 14.1: Definition einer abhängigen Eigenschaft *Durchmesser* für ein Objekt *Kreis*

C#

```
using System.Windows;

namespace BSP_Controls
{
```

Wir definieren eine neue Klasse *Kreis* und leiten diese gleich von *DependencyObject* ab:

```
    class Kreis : DependencyObject
    {
```

Hier die eigentliche Definition der abhängigen Eigenschaft (übergeben werden der Name, der Datentyp, das abhängige Objekt, optional die Metadaten, d h. der Defaultwert und eine Callback-Methode):

```
        public static readonly DependencyProperty DurchmesserProperty =
             DependencyProperty.Register("Durchmesser", typeof(double), typeof(Kreis),
                                         new FrameworkPropertyMetadata(11.1,
                                         new
PropertyChangedCallback(OnDurchmesserChanged)));
```

Wie Sie sehen, handelt es sich nicht mehr um ein privates Feld, vielmehr wird die bisher übliche Kapselung aufgegeben, die Verwaltung des Zustands wird von WPF übernommen, die Instanz meldet seine "lokalen Speicher" nur noch an (*Register*).

Die folgende Definition ist nur noch die allgemeine Schnittstelle nach außen, wie Sie es auch von den normalen Eigenschaften kennen:

```
        public double Durchmesser
        {
            get { return (double)(GetValue(DurchmesserProperty)); }
            set { SetValue(DurchmesserProperty, value); }
        }
```

Hier eine Callback-Methode, mit der eine Wertänderung überwacht werden kann:

```
        private static void OnDurchmesserChanged(DependencyObject obj,
            DependencyPropertyChangedEventArgs args)
        {
            MessageBox.Show( (obj as Kreis).Durchmesser.ToString());
        }
    }
}
```

14.1.2 Angehängte Eigenschaften (Attached Properties)

Mit den angehängten Eigenschaften (*Attached Properties*), einer speziellen Form der Dependency Properties, wird der Versuch unternommen, die Flut an WPF-Element-Eigenschaften etwas einzudämmen. Das Problem: Wird ein Element/Control in einem Container platziert, hängt beispielsweise dessen Position vom umgebenden Container (*Grid, Canvas, DockPanel* etc.) ab. Für jede Art von Container werden aber andere Eigenschaften zur Positionierung benötigt. Aus diesem Grund stellen die übergeordneten Elemente die zum Positionieren nötigen Eigenschaften zur Verfügung (*Canvas.Top*, *Canvas.Left*, *DockPanel.Dock*, *Grid.Column* ...), das eingelagerte Element nutzt diese lediglich in seinem Kontext. Der Vorteil: nur wenn sich beispielsweise ein *Button* in einem *Canvas* befindet, werden auch die Eigenschaften *Canvas.Top, Canvas.Left* bereitgestellt und verwendet.

BEISPIEL 14.2: Positionieren einer Schaltfläche in einem *Canvas*-Control mit Attached Properties

XAML
```
<Canvas Name="canvas1" >
  <Button Canvas.Left="74" Canvas.Top="70" Height="45" Name="button1"
          Width="89">Beschriftung
  </Button>
</Canvas>
```

14.2 Einsatz von Ressourcen

Bevor wird uns im Weiteren mit Styles etc. beschäftigen, müssen wir zunächst noch einen Blick auf die Ressourcen-Verwaltung in WPF-Anwendungen werfen, da diese die Voraussetzung für die Zuordnung darstellt.

14.2.1 Was sind eigentlich Ressourcen?

In WPF-Anwendungen zählen nicht nur Grafiken, Strings, Sprachinformationen oder beliebige binäre Informationen zu den Ressourcen, sondern auch die Beschreibung von Styles, Füllmustern oder sogar ganzer Controls.

Eine Ressource besteht immer aus einem eindeutigen Schlüssel (Key) und dem eigentlichen Content, der jederzeit austauschbar ist, da Bezüge auf die Ressource immer nur den Schlüssel verwenden.

14.2.2 Wo können Ressourcen gespeichert werden?

Eine Möglichkeit, Ressourcen in Ihrer Anwendung abzulegen, haben Sie sicher ganz unbewusst schon zur Kenntnis genommen. Die Rede ist von der Datei *App.xaml*, in der bereits ein *Resources*-Abschnitt vordefiniert ist:

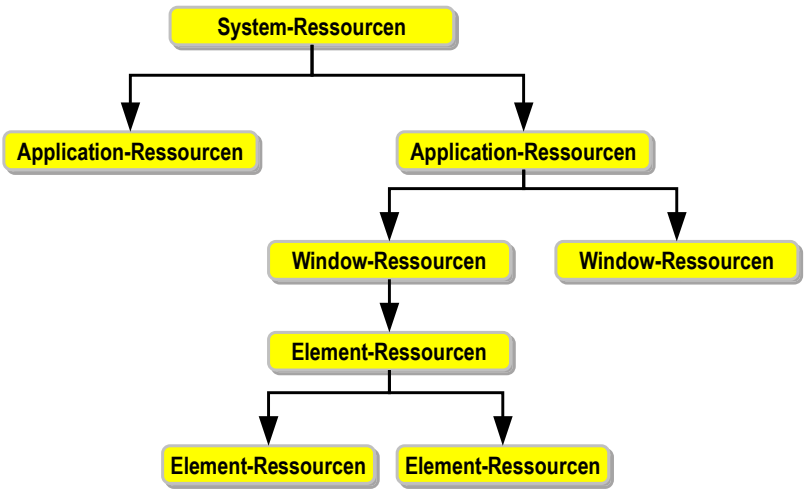
```
<Application x:Class="BSP_Controls.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    StartupUri="Test.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Hierbei handelt es sich um anwendungsweit verfügbare Ressourcen, die allen Windows/Pages und deren Elementen zur Verfügung stehen.

Neben diesen Ressourcen können Sie auch jedem Element und jedem Window eigene Ressourcen zuordnen, zusätzlich sind auch so genannte System-Ressourcen (z.B. Systemfarben) verfügbar.

Die folgende Abbildung zeigt die mögliche Hierarchie von Ressourcen:



HINWEIS: Wird eine Ressource referenziert und damit auch gesucht, beginnt die Suche immer beim aktuellen Element. Wird die Ressource hier nicht gefunden, wird im übergeordneten Element (Container → Window → Application → System) gesucht. Damit ist auch klar, dass untergeordnete Elemente gleichnamige Ressourcen von übergeordneten Elementen überschreiben können. Innerhalb einer Hierarchieebene ist dies nicht möglich, da es sich in diesem Fall nicht um einen eindeutigen Schlüssel handelt.

Im XAML-Quellcode stellt sich die Definition von *Resources*-Abschnitten wie folgt dar:

Die Window-Ressourcen:

```
<Window>
  <Window.Resources>
    ...
  </Window.Resources>

  <StackPanel>
```

Die Ressourcen eines Containers:

```
    <StackPanel.Resources>
      ...
    </StackPanel.Resources>
...
    <Button>
```

Die Ressourcen eines Elements:

```
      <Button.Resources>
      ...
      </Button.Resources>
    </Button>
  </StackPanel>
</Window>
```

14.2.3 Wie definiere ich eine Ressource?

Haben Sie sich dafür entschieden, auf welcher Ebene Sie die Ressource definieren, können Sie zur Tat schreiten:

- Zunächst müssen Sie sich über die Art der Ressource im Klaren sein, diese bestimmt den Elementnamen (z.B. ein *ImageBrush,* mit dem ein Hintergrund festgelegt werden kann).
- Neben dieser Information müssen Sie sich noch für einen eindeutigen Schlüssel entscheiden, über den die Ressource angesprochen werden soll.
- Last but not least, müssen Sie auch noch die eigentlichen Informationen zur Ressource (beim *ImageBrush* ist dies die *ImageSource)* festlegen.

BEISPIEL 14.3: Erzeugen und Verwenden einer Ressource auf Fenster-Ebene

XAML

```
<Window x:Class="Konzepte.Ressourcen_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Ressourcen_Bsp" Height="329" Width="464">
 <Window.Resources>
   <ImageBrush x:Key="bck" ImageSource="back.jpg" />
 </Window.Resources>
...
```

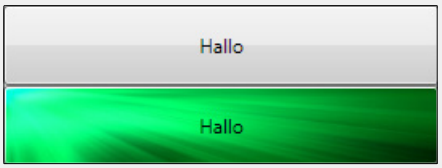
Die Verwendung:

```
 <StackPanel>
   <Button Background="{StaticResource bck}">Hallo</Button>
 </StackPanel>
...
```

BEISPIEL 14.3: Erzeugen und Verwenden einer Ressource auf Fenster-Ebene

Ergebnis

Die Unterschiede in der Darstellung dürften Ihnen sicher auffallen:



HINWEIS: Beim Einsatz von (statischen) Ressourcen ist es wichtig, dass diese **vor** der Verwendung definiert wurden, andernfalls kann die Ressource nicht gefunden werden. Zusätzlich müssen Sie auf die Schreibweise achten, hier wird die Groß-/Kleinschreibung berücksichtigt!

Alternativ können Sie bei der Verwendung der Ressource auch die folgende Syntax nutzen:

```
<Button Height="50">
  <Button.Background>
    <StaticResource ResourceKey="bck" />
  </Button.Background>
  Hallo
</Button>
```

14.2.4 Statische und dynamische Ressourcen

Wie Sie im vorhergehenden Beispiel bereits gesehen haben, können Ressourcen statisch, d.h unveränderlich, über den Schlüssel zugeordnet werden:

```
<StackPanel>
  <Button Background="{StaticResource bck}">Hallo</Button>
</StackPanel>
```

Voraussetzung war das vorherige Definieren dieser Ressource. Spätere Änderungen an der Ressource werden nicht registriert und haben keine Auswirkung auf das verwendende Element.

Neben dieser Variante besteht auch die Möglichkeit, Ressourcen dynamisch festzulegen. In diesem Fall können Sie die Ressourcen zur Laufzeit zuordnen bzw. bereits vorhandene Ressourcen einfach austauschen.

BEISPIEL 14.4: Verwendung einer noch nicht definierten dynamischen Ressource

XAML

Zunächst die Zuordnung von Eigenschaft und Ressource:

```
<Button Height="50" Background="{DynamicResource bck1}"
        Click="Button_Click">Hallo</Button>
```

BEISPIEL 14.4: Verwendung einer noch nicht definierten dynamischen Ressource

C#

Mit dem Klick auf die Schaltfläche wollen wir eine Ressource zuordnen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

Neuen *ImageBrush* erzeugen:

```
    ImageBrush imgbck = new ImageBrush();
```

Eine Bitmap zuweisen:

```
    imgbck.ImageSource = new BitmapImage(new
Uri("pack://application:,,,/back2.jpg"));
```

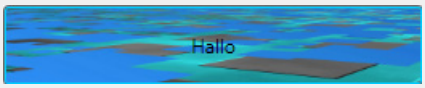
Die Ressource erzeugen (Window-Ressource):

```
    this.Resources.Add("bck1", imgbck);
}
```

Ergebnis

Vor dem Klick auf die Schaltfläche:

Nach dem Klick auf die Schaltfläche:



Und warum verwenden wir nicht einfach immer dynamische Ressourcen? Da die Verwaltung dynamischer Ressourcen deutlich aufwändiger ist, würden hier unnötigerweise Systemressourcen verschwendet werden.

14.2.5 Wie werden Ressourcen adressiert?

Möchten Sie auf eingebundene Ressourcen (z.B. Grafiken) Ihrer Anwendung zugreifen, müssen Sie beispielsweise bei der Zuordnung von Grafiken per Code eine bestimmte Syntax einhalten, andernfalls wird die Ressource an der falschen Stelle gesucht und dann wohl auch nicht gefunden.

Die dazu erforderliche URI können Sie absolut, d h. mit voller Pfadangabe, inklusive der aktuellen Assembly angeben oder relativ zur aktuellen Assembly.

BEISPIEL 14.5: Absolute Pfadangabe (Bild liegt in der Projekt-Root, Buildvorgang= Resource)

C#

```
Uri uri = new Uri("pack://application:,,,/Bild.jpg");
```

BEISPIEL 14.6: Absolute Pfadangabe (das Bild liegt im Unterverzeichnis \\Images des aktuellen Projekts, Buildvorgang=Resource)

C#

```
Uri uri = new Uri("pack://application:,,,/Images/Bild.jpg");
```

BEISPIEL 14.7: Relative Pfadangabe (das Bild liegt im gleichen Verzeichnis wie die Assembly)

C#
```
Uri uri = new Uri(@".\Back3.jpg", UriKind.Relative);
```

BEISPIEL 14.8: Relative Pfadangabe (das Bild liegt im relativen Unterverzeichnis \\Images der Assembly)

C#
```
Uri uri = new Uri(@".\Images\Back3.jpg", UriKind.Relative)
```

Neben den gezeigten Möglichkeiten können Sie unter anderem auch auf eingebundene Assemblies zugreifen, eine (mehr oder weniger fehlerhafte) Dokumentation zu dieser Gesamtthematik finden Sie unter der folgenden Adresse:

LINK: http://msdn.microsoft.com/de-de/library/aa970069.aspx

14.2.6 System-Ressourcen einbinden

Neben den in Ihrer Anwendung definierten Ressourcen können Sie auch System-Ressourcen verwenden. Die Einbindung erfolgt entweder statisch oder dynamisch, im erstem Fall reagiert die Anwendung jedoch nicht auf aktuelle Änderungen an den Systemeinstellungen.

BEISPIEL 14.9: Einbinden von System-Ressourcen

XAML

Anzeige der *VirtualScreenWidth* in einem *Label* (Ressourcenabfrage):

```
<Label Content="{StaticResource {x:Static SystemParameters.VirtualScreenWidthKey}}"/>
```

Anzeige, ob User-Interface-Effekte aktiviert sind (Wertabfrage):

```
<CheckBox IsChecked="{x:Static SystemParameters.UIEffects}" Content="UIEffects" />
```

14.3 Das WPF-Ereignis-Modell

Nachdem wir in den bisherigen Beispielen recht sparsam mit der Verwendung von Event-Handlern umgegangen sind, wollen wir uns jetzt diesem Thema etwas intensiver widmen.

14.3.1 Einführung

Sicher haben Sie auch schon mehr oder weniger unbewusst von den Ereignissen in WPF Gebrauch gemacht. Nach dem Klick, z.B. auf eine Schaltfläche, wird der entsprechende Ereignis-Handler von Visual Studio bereitgestellt.

BEISPIEL 14.10: *Button* mit zugehöriger Ereignisbehandlung

XAML
```
<StackPanel>
  <Button Content="Klick mich!" Click="Button_Click"/>
</StackPanel>
```

BEISPIEL 14.10: *Button* mit zugehöriger Ereignisbehandlung

C#

Der Ereignis-Handler:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo");
}
```

So weit – so gut, das kennen Sie sicher auch schon von der Programmierung in Win32-/Windows Forms-Anwendungen so. Doch was passiert, wenn Sie in WPF eine Schaltfläche aus einzelnen Elementen "zusammenbasteln"?

BEISPIEL 14.11: Button mit Text und Grafik

XAML

```
<Button Click="Button_Click_1">
  <StackPanel Orientation="Horizontal" Margin="10">
    <Image Source="Images/Flash.png" Width="56" Height="46" />
    <TextBlock VerticalAlignment="Center">Klick mich!</TextBlock>
  </StackPanel>
</Button>
```

Ergebnis

Rein intuitiv haben Sie sicher auch das *Click*-Ereignis dem Button zugeordnet. Doch warum sollte das eigentlich funktionieren? Was passiert, wenn der Nutzer auf das *Image* oder den *TextBlock* klickt, zusätzlich befindet sich "darunter" ja noch das *StackPanel.*

Das Zauberwort heißt hier "Routed Events", ein Verfahren, um auftretende Ereignisse in der Element-Hierarchie weiterzugeben.

14.3.2 Routed Events

WPF unterscheidet bei den Routed Events zwei verschiedene Varianten, die Sie als Programmierer auch auseinander halten sollten:

- **Tunneling Events**
  Ausgehend vom Wurzelelement (*Window/Page)* werden die Ereignisse bis zum auslösenden Element weitergereicht. Diese Events werden **vor** den zugehörigen Bubbling Events ausgelöst.
- **Bubbling Events**
  Ausgehend vom aktivierten Element werden die Ereignisse zum jeweils übergeordneten Element weitergereicht, d h. im Endeffekt bis zum *Window* oder zur *Page*.

HINWEIS: Tunneling Events sind durch den Vorsatz "Preview..." gekennzeichnet, Bubbling Events verzichten auf einen Vorsatz.

BEISPIEL 14.12: Ablauf der Ereigniskette für einen Mausklick mit der linken Taste

XAML

```
<Window x:Class="Konzepte.Ereignisse"
...
    Title="Ereignisse" MouseLeftButtonDown="Window_MouseLeftButtonDown"
                       PreviewMouseLeftButtonDown="Window_PreviewMouseLeftButtonDown">
   <Button MouseLeftButtonDown="Button_MouseLeftButtonDown"
           PreviewMouseLeftButtonDown="Button_PreviewMouseLeftButtonDown">
     <StackPanel Orientation="Horizontal" Margin="10"
                 MouseLeftButtonDown="StackPanel_MouseLeftButtonDown"
                 PreviewMouseLeftButtonDown="StackPanel_PreviewMouseLeftButtonDown">
       <Image Source="Images/Flash.png" Width="56" Height="46"
              MouseLeftButtonDown="Image_MouseLeftButtonDown"
              PreviewMouseLeftButtonDown="Image_PreviewMouseLeftButtonDown" />
       <TextBlock VerticalAlignment="Center"
              MouseLeftButtonDown="TextBlock_MouseLeftButtonDown"
              PreviewMouseLeftButtonDown="TextBlock_PreviewMouseLeftButtonDown">Klick mich!
       </TextBlock>
     </StackPanel>
   </Button>
</Window>
```

Ergebnis

Die Ereigniskette nach einem Klick auf das *Image*:

1. Window: *PreviewMouseLeftButtonDown*
2. Button: *PreviewMouseLeftButtonDown*
3. StackPanel: *PreviewMouseLeftButtonDown*
4. Image: *PreviewMouseLeftButtonDown*
5. Image: *MouseLeftButtonDown*
6. StackPanel: *MouseLeftButtonDown*

Wie Sie sehen können, wird zunächst die komplette Tunneling-Ereigniskette durchlaufen, nachfolgend die Bubbeling-Events[1].

Im Normalfall werden Sie wohl nur Bubbling-Events verwenden, Tunneling-Events nutzen Sie beispielsweise, um Ereignisse bzw. deren Weiterleitung zu blockieren.

[1] Der Button löst ein *Click*-Ereignis aus, die dazu nötige Logik verhindert das Auslösen entsprechender *MouseLeftButtonDown*-Ereignisse, deshalb ist hier die Ereigniskette zu Ende.

BEISPIEL 14.13: Das Weiterleiten der Ereignisse verhindern

C#

Wir werten gleich das erste Ereignis aus (das Tunneling Event beginnt an der Root, d h. dem Window):

```
private void Window_PreviewMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
```

Ereignis behandelt (Klappe zu, Affe tot):

```
    e.Handled = true;
    List1.Items.Add("Window_PreviewMouseLeftButtonDown");
}
```

BEISPIEL 14.14: Das auslösende Element bestimmen

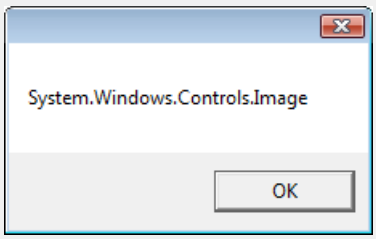
C#

Den Ursprung für ein Ereignis können Sie mit dem *...EventArgs.Source*-Parameter bestimmen:

```
private void Window_PreviewMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show(e.Source.ToString());
    List1.Items.Add("Window_PreviewMouseLeftButtonDown");
}
```

Ergebnis

Beim Klick auf das Image wird auch dieses als *Source* übermittelt:



14.3.3 Direkte Events

Auch diese Form der Ereignisse ist nach wie vor präsent, hierbei handelt es sich um die ganz normalen .NET-Ereignisse, wie Sie auch in Windows-Forms-Anwendungen auftreten.

Direkte Events werden eingesetzt, wenn die Verwendung von Bubbling oder Tunneling keinen Sinn macht, beispielsweise beim *MouseLeave*-Ereignis, das sehr objektbezogen ausgelöst wird.

Sie erkennen diese Ereignisse an einem fehlenden *Preview...*-Pendant.

14.4 Verwendung von Commands

Im Zusammenhang mit der Entwicklung von Anwendungen ist es Ihnen sicher auch schon passiert, dass Sie eine Menüfunktion zum x-ten Male neu programmiert haben. Das Gleiche trifft sicher ebenfalls auf den Toolbar zu. Das Prozedere ist doch immer gleich:

1. Methode mit der eigentlichen Logik erstellen
2. Menüpunkt erstellen
3. Menüpunkt Tastenkürzel zuweisen, Tastaturabfrage implementieren
4. Menüpunkt Ereignismethode zuweisen und Methode von 1. aufrufen
5. Toolbar-Button bereitstellen
6. Toolbar-Button Ereignismethode zuweisen und Methode von 1. aufrufen
7. Logik für das Sperren von 2. und 5. erstellen, wenn die Funktion nicht zur Verfügung steht.

14.4.1 Einführung zu Commands

In WPF-Anwendungen können Sie diesen Aufwand wesentlich verringern, indem Sie entweder die von WPF vordefinierten Commands verwenden, oder indem Sie selbst eigene Commands implementieren.

Die neue Vorgehensweise (vordefinierte Commands, z.B. Einfügen in die Zwischenablage) im Vergleich zum bisherigen Vorgehen:

1. entfällt, da für viele Controls bereits implementiert
2. **Menüpunkt erstellen und Command zuweisen**
3. entfällt, da per Command automatisch zugewiesen
4. entfällt, da per Command automatisch zugewiesen
5. **Toolbar-Button bereitstellen und Command zuweisen**
6. entfällt, da per Command automatisch zugewiesen
7. entfällt, da Command diese Logik bereitstellt.

Das sieht doch schon wesentlich freundlicher aus als die mühsame und fehleranfällige erste Variante. Ähnlich einfach gestaltet sich die Wiederverwendung von selbst erstellten Kommandos, wenn Sie diese bereits einmal erstellt haben.

14.4.2 Verwendung vordefinierter Commands

Statt vieler Worte wollen wir Ihnen zunächst an einem Beispiel die Vorgehensweise bei der Verwendung von Commands demonstrieren.

BEISPIEL 14.15: Programmieren der Funktionen "Kopieren und Einfügen" für ein Formular mit zwei TextBoxen

XAML

Der XAML-Code der Oberfläche:

```
<Window x:Class="Konzepte.Commands_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Commands_Bsp" Height="373" Width="411">
 <DockPanel>
```

Die Menüdefinition:

```
   <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
...
     <MenuItem Header="_Bearbeiten">
```

Hier werden die beiden Menüpunkte "Kopieren" und "Einfügen" definiert:

```
       <MenuItem Command="ApplicationCommands.Copy"/>
       <MenuItem Command="ApplicationCommands.Paste"/>
```

Sie können auf die Angabe des Headers sowie der Tastenkürzel verzichten, dies stellt die obige *Command*-Definition automatisch zur Verfügung[1].

```
     </MenuItem>
   </Menu>
   <ToolBarTray DockPanel.Dock="Top">
     <ToolBar>
```

Hier findet sich bereits die Definition für die *ToolBar*-Buttons, auch in diesem Fall genügt die Zuordnung der *Command*-Eigenschaft:

```
       <Button Width="30" Height="30" Command="ApplicationCommands.Copy">
         <Image Source="Images/editcopy.png"/>
       </Button>
       <Button Width="30" Height="30" Command="ApplicationCommands.Paste">
         <Image Source="Images/editpaste.png"/>
       </Button>
     </ToolBar>
   </ToolBarTray>
   <StackPanel>
```

Hier noch die beiden *TextBox*en, für die die Funktionen implementiert werden:

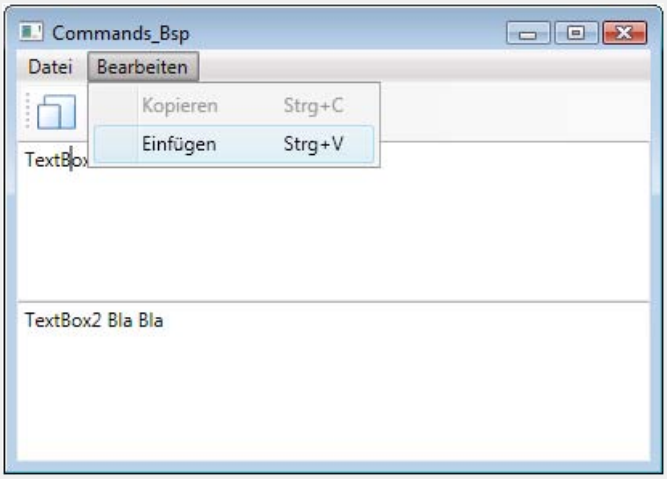
```
     <TextBox Name="txt1" Height="100">TextBox1 Bla Bla</TextBox>
     <TextBox Name="txt2" Height="100">TextBox2 Bla Bla</TextBox>
...
   </StackPanel>
 </DockPanel>
</Window>
```

[1] Geben Sie trotzdem einen Wert an, überschreibt dies die vom Command bereitgestellten Werte.

BEISPIEL 14.15: Programmieren der Funktionen "Kopieren und Einfügen" für ein Formular mit zwei TextBoxen

Ergebnis

Und wo bleibt der Code? Kurze Antwort: Ihr Programm ist an dieser Stelle bereits fertig. Nach dem Start können Sie sich von der Funktionstüchtigkeit überzeugen:



Beide Menüpunkte verfügen über eine Beschriftung und ein Tastenkürzel, befindet sich Text in der Zwischenablage, ist das Menü *Einfügen* aktiviert, andernfalls ist der Menüpunkt gesperrt. Der Menüpunkt *Kopieren* ist nur aktiv, wenn Sie in einer der *TextBox*en Text markieren und diese *TextBox* den Eingabefokus besitzt.

Nicht schlecht, wenn Sie dies mit der konventionellen Variante vergleichen, bei der Sie sicher wesentlich mehr Code produziert hätten.

14.4.3 Das Ziel des Commands

Doch gerade bei obigem Beispiel wird sicher bei manchem die Frage aufkommen, wohin denn eigentlich der Text eingefügt wird, haben wir doch zwei Textfelder. Die Frage ist sicher berechtigt, aber im obigen Beispiel noch recht einfach lösbar: Ziel des jeweils gewählten Commands ist standardmäßig immer das gerade aktive Control, d h. die *TextBox*, die den Eingabefokus besitzt.

Schwieriger wird es, wenn die Zwischenablageinhalte gezielt in ein bestimmtes Control eingefügt werden sollen. In diesem Fall müssen Sie neben der *Command*- auch die *CommandTarget*-Eigenschaft definieren. Allerdings genügt in diesem Fall nicht die reine Angabe des Element-Namens, Sie müssen die Binding-Syntax verwenden.

BEISPIEL 14.16: Zwischenablageinhalte sollen immer in *TextBox2* eingefügt werden

XAML

Wir fügen dem betreffenden Menüpunkt ein *CommandTarget* hinzu:

```
<MenuItem Header="_Bearbeiten">
```

BEISPIEL 14.16: Zwischenablageinhalte sollen immer in *TextBox2* eingefügt werden

```
    <MenuItem Command="ApplicationCommands.Copy"/>
    <MenuItem Command="ApplicationCommands.Paste" CommandTarget="{Binding ElementName=txt2}"/>
  </MenuItem>
...
```

Ergebnis

Starten Sie jetzt das Programm, ist es egal, welches Control den Fokus besitzt, drücken Sie die Tastenkombination *Strg+V* oder wählen Sie den entsprechenden Menüpunkt, wird der Text immer in die *TextBox2* eingefügt.

HINWEIS: Diese Vorgehensweise müssen Sie auch wählen, wenn Sie Commands von einzelnen Schaltflächen aus starten wollen. Da diese den Fokus erhalten können, würde nie klar sein, welches Ziel die Aktion haben soll.

BEISPIEL 14.17: "Freistehender" *Button* für das Einfügen des Zwischenablageinhalts

XAML

```
<Button Command="ApplicationCommands.Paste" CommandTarget="{Binding ElementName=txt2}">
    Paste (TextBox 2)
</Button>
```

14.4.4 Welche vordefinierten Commands stehen zur Verfügung?

WPF bietet bereits "ab Werk" eine ganze Reihe von Commands, die in der täglichen Programmierpraxis immer wieder anfallen. Diese Commands gliedern sich in die folgenden Gruppen:

- ApplicationCommands
  (z.B. *Cut, Past, Help, New, Print, Save, Stop, Undo*)
- ComponentCommands
  (z.B. *ExtendSelectionLeft*, *MoveLeft*)
- NavigationCommands
  (z.B. *FirstPage, GotoPage, Refresh*, *Search*)
- MediaCommands
  (z.B. Play, *Pause, FastForward*, *IncreaseVolume*)
- EditingCommands
  (z.B. *Delete*, *MoveUpByLine, ToggleBold*[1])

HINWEIS: Doch Achtung: Nicht das Command an sich stellt die Logik, z.B. für das Einfügen von Zwischenablageinhalten, zur Verfügung, sondern das jeweilige Ziel-Objekt, d h. ein spezifisches Control wie *TextBox* oder *Image*.

[1] siehe dazu auch ab Seite 676 (*RichTextBox*)

Ob, und wenn ja welche, Commands implementiert sind, müssen Sie von Fall zu Fall ausprobieren. Sehr umfassend ist z.B. die Unterstützung bei *TextBox* und *RichTextBox* sowie beim *MediaElement*.

14.4.5 Commands an Ereignismethoden binden

Wie schon erwähnt, implementiert nicht jedes Control alle verfügbaren Commands. So steht zwar ein *ApplicationCommand.Open* (d h. Beschriftung und Tastaturkürzel) zur Verfügung, im Normalfall passiert allerdings nichts, da kein Control eine entsprechende Logik implementiert hat, was bei derart komplexen Abläufen sicher auch nicht möglich ist.

Aus diesem Grund besteht die Möglichkeit, einem Command eine entsprechenden Ereignisbehandlung per *CommandBinding* zuzuordnen. Zwei Ereignisse sind hier von zentraler Bedeutung:

- *Execute (*die eigentlich auszuführende Logik) und
- *CanExecute* (eine Abfrage, ob die Funktion überhaupt zur Verfügung steht)

BEISPIEL 14.18: Implementieren des *ApplicationCommand.Open*

XAML

Nutzen Sie für die Zuordnung der beiden Ereignismethoden am besten das Wurzel-Element (*Windows/Page*):

```
<Window x:Class="Konzepte.Commands_Bsp"
...
    Title="Commands_Bsp" Height="373" Width="411">
 <Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Open"
                   Executed="OpenCmdExecuted" CanExecute="OpenCmdCanExecute"/>
 </Window.CommandBindings>
...
```

C#

Die beiden zugehörigen Ereignismethoden aus der Klassendefinition des Windows:

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Was soll ich öffnen????");
    // hier steht die eigentliche Logik
}
```

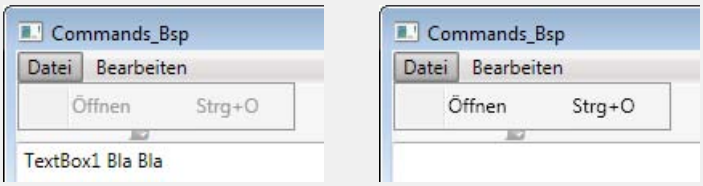
Nur wenn die erste *TextBox* auch leer ist, ist das Öffnen neuer Dateien möglich:

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (txt1.Text.Length == 0)
       e.CanExecute = true;
    else
       e.CanExecute = false;
}
```

BEISPIEL 14.18: Implementieren des *ApplicationCommand.Open*

Ergebnis

Ein Test zur Laufzeit zeigt, dass die entsprechende Schaltfläche nur freigegebenen ist, wenn die *TextBox* leer ist:



Klicken Sie auf den obigen Menüpunkt oder verwenden Sie die Tastenkombination *Strg+O*, wird unsere MessageBox aus der Methode *OpenCmdExecuted* ausgeführt.

HINWEIS: Selbstverständlich können Sie CommandBinding auch per Code realisieren, wie das folgende Beispiel zeigt.

BEISPIEL 14.19: Ereignismethode per Code zuweisen

C#

```
public Commands_Bsp()
{
    InitializeComponent();
    CommandBinding cmdOpen = new CommandBinding(ApplicationCommands.Open);
    cmdOpen.Executed += new ExecutedRoutedEventHandler(cmdOpen_Executed);
}

void cmdOpen_Executed(object sender, ExecutedRoutedEventArgs e)
{ ... }
```

14.4.6 Wie kann ich ein Command per Code auslösen?

Neben der bereits beschriebenen Variante, Commands per Zuordnung im XAML-Code auszulösen, besteht auch die Möglichkeit, diese direkt mit der *Execute*-Methode aufzurufen.

BEISPIEL 14.20: Direkter Aufruf eines Commands in einer Ereignismethode

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ApplicationCommands.Paste.Execute(null, txt2);
}
```

HINWEIS: Im zweiten Parameter übergeben Sie das *CommandTarget*.

14.4.7 Command-Ausführung verhindern

Nicht immer und zu jeder Zeit können Kommandos einfach ausgeführt werden. Unter bestimmten Bedingungen steht eine Funktion nicht zur Verfügung, in diesem Fall sollen die Menüpunkte/Schaltflächen abgeblendet sein, um den Anwender nicht unnötig zu verwirren.

Die Lösungsmöglichkeit haben wir Ihnen bereits beim Zuordnen von Ereignismethoden gezeigt. Im *...CanExecute*-Ereignis wird mit dem Parameter *e.CanExecute* entschieden, ob eine Aktion ausführbar ist oder nicht.

BEISPIEL 14.21: Command-Ausführung verhindern

C#
```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
      e.CanExecute = false;
}
```

14.5 Das WPF-Style-System

Mit den WPF-Styles kommen wir jetzt zu einem Thema, das sicher von ganz zentraler Bedeutung für die oberflächenorientierten WPF-Anwendungen ist und auch einen der wesentlichsten Unterschiede zu den konventionellen Windows-Anwendungen darstellt.

14.5.1 Übersicht

Doch worum geht es eigentlich? Sicher haben Sie nach der Lektüre der beiden vorhergehenden Kapitel festgestellt, dass WPF-Controls mit Bergen von Eigenschaften ausgestattet sind, die es möglich machen, fast alle Aspekte der Darstellung zu beeinflussen.

Doch gerade für aufwändige Oberflächen ergeben sich einige Fragen:

1. Wie kann ich mehr als einem Control ein spezifisches Aussehen zuweisen?
2. Wie kann ich das Aussehen unter bestimmten Bedingungen ändern?
3. Wie kann ich das grundsätzliche Aussehen eines Controls komplett ändern?
4. Wie kann ich einfache Animationen realisieren?

Für alle diese Fragen bietet WPF eine Antwort:

1. Verwendung von benannten oder Typ-Styles
2. Verwendung von Triggern
3. Verwendung von Templates
4. Verwendung von StoryBoards

HINWEIS: Im Rahmen dieses Abschnitts werden wir die o.g. Themen nur recht oberflächlich streifen, da dies eigentlich ein Hauptarbeitsgebiet für den Designer und nicht für den Programmierer ist[1]. Außerdem ist Visual Studio für einige der obigen Aufgaben das falsche Tool. Hier kommen Sie mit *Microsoft Expression Blend* (siehe Seite 781) wesentlich weiter, da der Designer teilweise komfortabler ist. Wir weisen Sie an den entsprechenden Stellen darauf hin.

14.5.2 Benannte Styles

In den bisherigen Beispielen haben wir die Eigenschaften jedes Controls einzeln gesetzt, was bei größeren Ansammlungen recht schnell zum Geduldsspiel ausarten kann. Denken Sie nur an unseren Taschenrechner aus dem WPF-Einführungskapitel, wo ca. zwanzig einzelne Tasten zu konfigurieren sind (Schrift, Randabstände, Farben etc.). Viel schöner wäre doch hier eine zentrale Vorschrift, wie ein derartiger Button auszusehen hat.

Genau diese Aufgabe übernimmt ein *benannter Style*. Dieser wird einmal definiert und kann dann per Key den einzelnen Elementen zugewiesen werden.

BEISPIEL 14.22: Alle Schaltflächen für den Taschenrechner sollen einen Randabstand von 2, eine fette Schrift und eine gelbe Hintergrundfarbe bekommen.

XAML

```
<Window x:Class="Konzepte.Taschenrechner_1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window2" Height="199" Width="388">
```

Oh, jetzt wird auch klar, warum wir uns in diesem Kapitel bereits mit Ressourcen beschäftigt haben. Hoffentlich haben Sie diesen Abschnitt nicht gelangweilt überblättert!

```
 <Window.Resources>
```

Wir definieren einen neuen Style und legen dessen *Key* fest (über diesen könne wir später auf den Style zugreifen bzw. auf diesen verweisen):

```
   <Style x:Key="myBtnStyle">
```

Innerhalb des *Style*-Elements können Sie per *Setter* die gewünschten Eigenschaften beeinflussen:

```
     <Setter Property="Control.Margin" Value="2" />
     <Setter Property="Control.Background" Value="Yellow" />
     <Setter Property="Control.FontWeight" Value="UltraBold" />
```

Legen Sie jeweils *Property* und *Value* fest.

```
   </Style>
 </Window.Resources>
```

[1] Hier ist die Borderline zwischen Designer und Programmierer, mit teilweise diffusem Verlauf ...

BEISPIEL 14.22: Alle Schaltflächen für den Taschenrechner sollen einen Randabstand von 2, eine fette Schrift und eine gelbe Hintergrundfarbe bekommen.

Und jetzt kommt unser neuer Style zum Einsatz (setzen Sie diesen nur bei den Ziffertasten):

```
<UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
  <Button  Name="button1"  Style="{StaticResource myBtnStyle}">7</Button>
  <Button  Name="button2"  Style="{StaticResource myBtnStyle}">8</Button>
  <Button  Name="button3"  Style="{StaticResource myBtnStyle}">9</Button>
...
```

Ergebnis

Schon im Designer dürfte sich jetzt etwas getan haben:

Was passiert eigentlich, wenn wir den Style einer *TextBox* zuweisen? Probieren Sie es ruhig aus, es kann nichts passieren. Vielleicht sind Sie überrascht, aber auch die *TextBox* wird nach dieser Aktion im grässlichen gelben Outfit erscheinen und fette Schrift anzeigen.

Warum dies so ist? Ganz einfach, auch die *TextBox* verfügt über die aufgeführten Eigenschaften und übernimmt diese automatisch vom Style.

14.5.3 Typ-Styles

Ein "fauler" Programmierer wird immer einen einfacheren Weg suchen, und so ist es sicher mühsam, den Style jedem einzelnen Button einzeln zuzuweisen, zumal die Syntax auch recht umfangreich ist. Aus diesem Grund gibt es noch eine zweite Art von Styles, die nicht über einen Key, sondern indirekt über den Klassennamen zugeordnet werden.

Der Vorteil: Alle WPF-Elemente, die Instanzen dieser Klasse sind, übernehmen automatisch diesen Style, ohne dass dies explizit angegeben werden muss.

BEISPIEL 14.23: (Fortsetzung) Mit Ausnahme der Zifferntaste sollen alle Tasten einen grünen Hintergrund und weiße Schrift bekommen.

XAML

```
<Window.Resources>
```

Hier der benannte Style für die Zifferntasten:

```
  <Style x:Key="myBtnStyle">
...
```

BEISPIEL 14.23: (Fortsetzung) Mit Ausnahme der Zifferntaste sollen alle Tasten einen grünen Hintergrund und weiße Schrift bekommen.

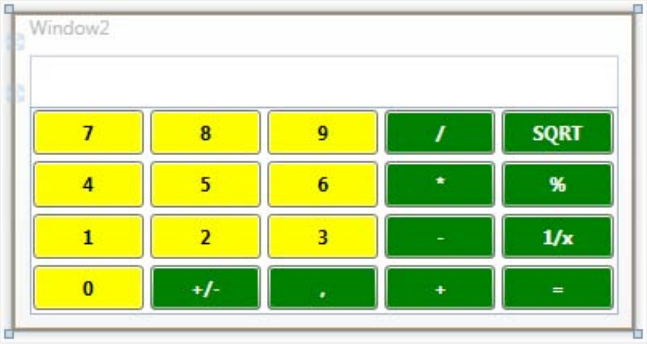
```
</Style>
```

Und hier der Default-Style für alle Elemente vom Typ *Button*:

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Margin" Value="2" />
  <Setter Property="Background" Value="Green" />
  <Setter Property="FontWeight" Value="UltraBold" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

Ergebnis

Ohne weitere Änderungen im XAML-Code dürfte sich jetzt bereits folgender Anblick bieten:



HINWEIS: Hätten Sie die Style-Definition in der Datei *App.xaml* eingefügt, würde es sich um einen anwendungsweiten Style handeln, alle Windows bzw. die enthaltenen Schaltflächen würden dieses Outfit bekommen.

Doch was, wenn sich auf Ihrem Formular zum Beispiel ein paar *ToggleButton*-Controls befinden? Diese sind von obiger Styledefinition nicht betroffen, handelt es sich doch um eine andere Klasse.

14.5.4 Styles anpassen und vererben

Es gibt immer wieder Ausnahmen von der Regel, und so kommen Sie meist nicht darum herum, den Style einzelner Controls speziell anzupassen. Drei Varianten bieten sich dazu an:

1. Sie überschreiben einzelne Attribute direkt im Element.
2. Sie ersetzen den Style auf einer niedrigeren Ebene (statt *Application* z.B. in einem *StackPanel*).
3. Sie vererben den Style und passen ihn unter neuem Namen an.

Styles anpassen (überschreiben)

Das Überschreiben von Styles ist eigentlich ganz intuitiv möglich, geben Sie den Key des Styles an oder nutzen Sie einen Typ-Style wie bisher. Gleichzeitig erweitern Sie die Attribut-Liste des betreffenden Elements, um die gewünschten Änderungen vorzunehmen.

BEISPIEL 14.24: (Fortsetzung) Die Taste "0" soll rot hinterlegt werden (der Style gibt gelb vor)

XAML
```
<Button  Name="button16" Style="{StaticResource myBtnStyle}" Background="Red" >0</Button>
```

Style ersetzen

Fällt ein komplettes Formular aus dem Rahmen, oder möchten Sie einzelne Elemente einer Gruppe (*StackPanel*, *Grid* etc.) mit einem angepassten Style versehen, können Sie auch den zentral gültigen Style ersetzen. Definieren Sie dazu einen neuen Ressource-Abschnitt und fügen Sie die neue Style-Definition in diesen ein.

BEISPIEL 14.25: (Fortsetzung) Ersetzen des zentralen Button-Styles

XAML
```
<Window x:Class="Konzepte.Taschenrechner_1"
...
```

Hier die übergreifende Definition:

```
 <Window.Resources>
   <Style TargetType="{x:Type Button}">
...
   </Style>
 </Window.Resources>
...
   <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
```

Hier wir der Style **ersetzt**, d.h., alle obigen Einstellungen gehen verloren:

```
    <UniformGrid.Resources>
      <Style TargetType="{x:Type Button}">
        <Setter Property="Margin" Value="2" />
        <Setter Property="Background" Value="Blue" />
        <Setter Property="FontWeight" Value="UltraBold" />
        <Setter Property="Foreground" Value="White" />
      </Style>
    </UniformGrid.Resources>
    ...
```

HINWEIS: Elemente, die sich in der Hierarchie oberhalb des *UniformGrids* befinden sind nicht von dieser Anpassung betroffen, hier gilt wieder die zentrale Version.

Styles vererben

Wer schreibfaul ist und beispielsweise nicht den kompletten Style austauschen will, kann diesen auch einfach vererben. Dazu wird in die Definition des Styles das Attribut *BasedOn* aufgenommen, das per Binding auf den Basisstyle verweist.

Ob Sie in diesem neuen Style bestehende Eigenschaften überschreiben (einfach erneut definieren) oder neue Eigenschaften hinzufügen, bleibt Ihnen überlassen.

BEISPIEL 14.26: Vererben eines Styles

XAML

```
<Window.Resources>
```

Das Original:

```
  <Style x:Key="myBtnStyle">
    <Setter Property="Control.Margin" Value="2" />
    <Setter Property="Control.Background" Value="Yellow" />
    <Setter Property="Control.FontWeight" Value="UltraBold" />
  </Style>
```

Der "Erbe" mit geringfügiger Änderung:

```
  <Style x:Key="myBtnStyle2" BasedOn="{StaticResource myBtnStyle}">
```

Hier wird eine Eigenschaft überschrieben:

```
    <Setter Property="Control.Margin" Value="5" />
```

Hier wird eine neue Eigenschaft gesetzt:

```
    <Setter Property="Control.FontStyle" Value="Italic" />
  </Style>
</Window.Resources>
```

Ergebnis

Links der neue Style, rechts der Basis-Style:

Styleänderung per Code

Ihr Programm ist nicht darauf angewiesen, immer den gleichen Style zu verwenden. So ist es problemlos möglich, auch zur Laufzeit einen Style per Code neu zu setzen (z.B. unter bestimmten Bedingungen).

BEISPIEL 14.27: Der Style von *button10* wird geändert

C#

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        { button10.Style = (Style)FindResource("myBtnStyle2");  }
```

HINWEIS: Den Style bzw. dessen Instanz finden Sie mit der per *FindResource* über dessen Key.

HINWEIS: Bevor Sie sich weiter in diese Thematik vertiefen, sollten Sie zunächst einen Blick auf den folgenden Abschnitt werfen, wahrscheinlich löst das Ihre Aufgabenstellung wesentlich eleganter.

14.6 Verwenden von Triggern

In unseren bisherigen Experimenten waren die vom Style vorgenommenen Änderungen immer statischer Natur, d h., einmal gesetzt, blieb die Optik immer gleich. Doch dies kann sicher nicht der Weisheit letzter Schluss sein.

Wenn jetzt in Ihnen der Programmierer wieder durchkommt und Sie an C# und Ereignisprozeduren denken, vergessen Sie es gleich wieder. Für (fast) alle Aufgabenstellungen ist auch hier XAML die beste Lösung.

Für die Reaktion auf Eigenschaftsänderungen, Ereignisse, Datenänderungen etc. können Sie in WPF-Anwendungen so genannte Trigger verwenden und damit zum Beispiel den Style ändern. Dabei sind Sie nicht auf einen einzelnen Trigger angewiesen, sondern Sie können der Trigger-Collection auch mehrere Ereignisse mit unterschiedlichen Bedingungen zuweisen.

Im Folgenden wollen wir uns die verschiedenen Triggerarten näher ansehen.

14.6.1 Eigenschaften-Trigger (Property Triggers)

Sicher kennen Sie auch das eine oder andere Programm, das exzessiv Gebrauch von diversen optischen Spielereien macht. Wird beispielsweise mit dem Mauskursor auf ein Control gezeigt, ändert sich dessen Rahmen oder die Hintergrundfarbe. Gleiches gilt für Eingabefelder die den Focus erhalten etc. In all diesen Fällen ändern sich Eigenschaften (*IsMouseOver*, *IsFocused*), auf die Sie bei der konventionellen Programmierung mit Ereignismethoden reagieren können. Mit Eigenschaften-Triggern können Sie Ihren C#-Quellcode von derartigem Ballast befreien und direkt per XAML-Code Änderungen am Control vornehmen.

BEISPIEL 14.28: Eine *TextBox* soll auf Änderungen von *IsMouseOver* und *IsFocused* mit Farbänderungen reagieren

XAML

```
<Window x:Class="Konzepte.Trigger_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Trigger_Bsp" Height="300" Width="300">
```

Einen Style für die *TextBox* erzeugen:

```
 <Window.Resources>
   <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
```

Der Außenabstand soll immer 2 betragen:

```
     <Setter Property="Margin" Value="2" />
```

BEISPIEL 14.28: Eine *TextBox* soll auf Änderungen von *IsMouseOver* und *IsFocused* mit Farbänderungen reagieren

Hier werden die Trigger definiert:

```
    <Style.Triggers>
```

Unter der Bedingung ...

```
      <Trigger Property="IsMouseOver" Value="True">
```

... wird die folgende Eigenschaft gesetzt:

```
        <Setter Property="Background" Value="Yellow" />
      </Trigger>
```

Unter der Bedingung ...

```
      <Trigger Property="IsFocused" Value="True">
```

... werden die folgenden Eigenschaften gesetzt:

```
        <Setter Property="Background" Value="Blue" />
        <Setter Property="Foreground" Value="White" />
      </Trigger>
    </Style.Triggers>
  </Style>
 </Window.Resources>
 <Grid>
  <StackPanel>
```

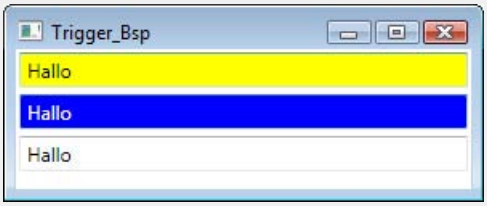
Hier verwenden wir den Style:

```
    <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
    <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
  </StackPanel>
 </Grid>
</Window>
```

Änderungen, die durch einen Trigger vorgenommen wurden, werden automatisch wieder rückgängig gemacht, wenn die Bedingung nicht mehr eingehalten wird (automatisches Wiederherstellen des Standardwertes).

Ergebnis

Die folgende Abbildung zeigt die Laufzeitansicht, die erste *TextBox* erfüllt die Bedingung *IsMouseOver=True*, die zweite *TextBox* hat den Eingabefokus und die dritte *TextBox* ist im Standardzustand:



Doch was, wenn Sie mehr als eine Bedingung benötigen? Auch das ist kein Problem, in diesem Fall erzeugen Sie einfach einen "multi-condition property trigger".

BEISPIEL 14.29: Der *TextBox*-Hintergrund soll rot werden, wenn die *TextBox* den Eingabefokus besitzt und wenn kein Text enthalten ist.

XAML

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
```

Einen Multi-Condition-Trigger definieren:

```
      <MultiTrigger>
        <MultiTrigger.Conditions>
```

Die beiden folgenden Bedingungen müssen zutreffen:

```
          <Condition Property="IsFocused" Value="True"/>
          <Condition Property="Text" Value="" />
        </MultiTrigger.Conditions>
```

Hier die Aktion:

```
        <Setter Property="Background" Value="Red" />
      </MultiTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

14.6.2 Ereignis-Trigger

Ereignis-Trigger werden durch bestimmte Ereignisse (vom Typ *RoutedEvent*) ausgelöst. Im Gegensatz zu den Eigenschaften-Triggern können Sie über derartige Trigger jedoch direkt keine Eigenschaften ändern, Sie können "lediglich" Animationen starten, die sich wiederum auf Eigenschaften auswirken.

HINWEIS: Ereignis-Trigger setzen geänderte Eigenschaften nicht wieder zurück, dafür sind **Sie** als Programmierer verantwortlich (z.B. im Pendant des betreffenden Ereignisses).

BEISPIEL 14.30: Wird der Mauskursor über einen Button bewegt, wird dieser transparent.

XAML

Zunächst die Style-Definition:

```
  <Style TargetType="{x:Type Button}">
    <Style.Triggers>
```

Hier kommt unser Ereignis-Trigger:

```
      <EventTrigger RoutedEvent="Button.MouseEnter">
```

BEISPIEL 14.30: Wird der Mauskursor über einen Button bewegt, wird dieser transparent.

Wir reagieren mit einer Aktion über die Hintergründe:

```
        <EventTrigger.Actions>
          <BeginStoryboard>
```

Hier die eigentliche Aktion, die Eigenschaft *Opacity* soll in 4 Sekunden von 1 auf 0.25 verringert werden:

```
            <Storyboard>
              <DoubleAnimation From="1" To="0.25" Duration="0:0:4"
                               Storyboard.TargetProperty="(Opacity)"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
```

Beim *MouseLeave*-Ereignis gehen wir den umgekehrten Weg und stellen die ursprüngliche Transparenz wieder her:

```
      <EventTrigger RoutedEvent="Button.MouseLeave">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation From="0.25" To="1" Duration="0:0:4"
                               Storyboard.TargetProperty="(Opacity)"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Style.Triggers>
  </Style>
```

14.6.3 Daten-Trigger

Mit diesen Triggern können Sie auf das Ändern beliebiger Eigenschaften reagieren, die Verbindung zu den entsprechenden Eigenschaften stellen Sie per Bindung her. Als Reaktion auf eine Eigenschaftsänderung können Sie, wie auch bei den Eigenschaften-Triggern, mit einem *Setter*-Element bestimmte Eigenschaften ändern.

HINWEIS: Die durch den Trigger geänderten Eigenschaften werden automatisch zurückgesetzt, wenn die Bedingung nicht mehr übereinstimmt.

BEISPIEL 14.31: Enthält die *TextBox* mehr als zehn Zeichen, wird sie grün eingefärbt.

XAML

```
<Window.Resources>
  <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
```

BEISPIEL 14.31: Enthält die *TextBox* mehr als zehn Zeichen, wird sie grün eingefärbt.

```
      <DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},
                  Path=Text.Length}" Value="10">
         <Setter Property="Background" Value="Green" />
      </DataTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

14.7 Einsatz von Templates

Im Folgenden möchten wir Sie zunächst mit ein paar Grundaussagen konfrontieren, bevor wir uns der Thematik "Templates" bzw. "Vorlagen" widmen:

- WPF-Controls haben prinzipiell keine Zeichenlogik, es handelt sich um Lookless Controls.
- WPF-Controls stellen lediglich eine Sammlung von Verhalten dar.

Spinnen die Autoren? Wo kommen denn sonst die ganzen optischen Spielereien her? Die Antwort auf dieses Paradoxon: Die Zeichenlogik eines Controls wird nur vom Layout/Styling bestimmt, jedes Control besitzt ein Standardaussehen (Default-Template), das komplett ersetzt werden kann.

Hier haben Sie es mit der Spielwiese der Designer zu tun, aus dem guten alten viereckigen Button kann ein gänzlich anderes Objekt werden, das jedoch nach wie vor das wesentliche Verhalten eines Buttons (Klick) besitzt. Aus Sicht des Programmierers kann dieser neue Button wie der Standard-Button verwendet werden. Damit ersparen Templates uns vielfach die Mühe, Controls umständlich abzuleiten und deren Zeichenlogik per C#-Code komplett neu zu implementieren.

HINWEIS: Im Gegensatz zu den Styles können Sie bei den Templates nicht nur die vorhandenen Eigenschaften beeinflussen, sondern das Control auch von Grund auf neu zusammenbauen.

14.7.1 Neues Template erstellen

Ein etwas komplexeres Beispiel soll die prinzipielle Vorgehensweise verdeutlichen, eine Komplettübersicht dieses Themas können wir Ihnen an dieser Stelle leider nicht geben.

BEISPIEL 14.32: Erzeugen und Verwenden eines Templates

XAML

Unsere Schaltflächen sollen ellipsenförmig sein und einen Farbverlauf aufweisen. Ist die Maus über dem Control, soll sich die Schriftstärke ändern. Ein Niederdrücken der Schaltfläche führt zu einem umgekehrten Farbverlauf im Control.

```
<Window.Resources>
```

Zunächst erstellen wir einen neuen Type-Style für *Button*-Elemente (Sie können auch einen benannten Style verwenden):

BEISPIEL 14.32: Erzeugen und Verwenden eines Templates

```
<Style TargetType="{x:Type Button}">
```

Hier greifen wir erstmals auf das Template zu, die Definition ist etwas verschachtelt, da es sich um eine recht komplexe Zuweisung handelt:

```
  <Setter Property="Template">
    <Setter.Value>
```

Der Eigenschaft *Template* wird ein *ControlTemplate* zugewiesen:

```
      <ControlTemplate TargetType="{x:Type Button}">
```

Für die innere Ausrichtung nutzen wir zunächst ein *Grid*:

```
        <Grid HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
ClipToBounds="False">
```

Und hier haben wir es schon mit der Optik zu tun, wir erzeugen eine Ellipse mit einem Farbverlauf (dies ist der Defaultzustand):

```
          <Ellipse Name="elli">
            <Ellipse.Fill>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1" >
                <LinearGradientBrush.GradientStops>
                  <GradientStop Color="#fff399" Offset="0.1"/>
                  <GradientStop Color="#ffe100" Offset="0.5"/>
                  <GradientStop Color="#feca00" Offset="0.9"/>
                </LinearGradientBrush.GradientStops>
              </LinearGradientBrush>
            </Ellipse.Fill>
          </Ellipse>
```

In unserem Button soll auch etwas angezeigt werden, dafür ist das *ContentPresenter*-Element verantwortlich:

```
          <ContentPresenter x:Name="PrimaryContent" HorizontalAlignment="Center"
                            VerticalAlignment="Center"
```

Den eigentlichen Inhalt holen wir uns wiederum vom ursprünglichen Element (dem *Button*), deshalb auch die etwas umständliche Bindung:

```
      Content="{Binding Path=Content, RelativeSource={RelativeSource TemplatedParent}}"
        />
      </Grid>
```

Nicht nur die Standardanzeige unseres Buttons wollen wir beeinflussen, sondern auch die Reaktion auf Maus und Klicken:

```
        <ControlTemplate.Triggers>
```

Es wird geklickt, d.h., wir zeichnen einen neuen Hintergrund. Dazu benötigen wir allerdings den Namen der Ellipse:

```
          <Trigger Property="Button.IsPressed" Value="True">
```

BEISPIEL 14.32: Erzeugen und Verwenden eines Templates

```
            <Setter Property="Fill" TargetName="elli" >
```

HINWEIS: Diese Namen sind nur innerhalb des Templates verwendbar!

```
              <Setter.Value>
                <LinearGradientBrush StartPoint="0,1" EndPoint="0,0" >
                  <LinearGradientBrush.GradientStops>
                    <GradientStop Color="#fff399" Offset="0.1"/>
                    <GradientStop Color="#ffe100" Offset="0.5"/>
                    <GradientStop Color="#feca00" Offset="0.9"/>
                  </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
              </Setter.Value>
            </Setter>
          </Trigger>
```

Die Maus wird über das Control bewegt, in diesem Fall wird lediglich die Schriftstärke geändert:

```
          <Trigger Property="Button.IsMouseOver" Value="True">
            <Setter Property="FontWeight" Value="Bold" />
          </Trigger>
```

Hier könnten Sie noch auf weitere Eigenschaftsänderungen reagieren ...

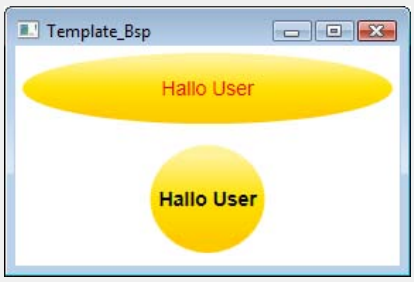
```
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
```

Last but not least, setzen wir noch ein paar Eigenschaften:

```
  <Setter Property="Foreground" Value="Black" />
  <Setter Property="FontFamily" Value="Arial" />
  <Setter Property="FontSize" Value="14" />
 </Style>
</Window.Resources>
```

Ergebnis

Und wie sieht nun unsere neue Schaltfläche aus? Hier die Antwort:



Doch wie können Sie innerhalb des Templates auf die ursprüngliche Definition von Eigenschaften zugreifen? Hier hilft Ihnen ebenfalls Binding weiter.

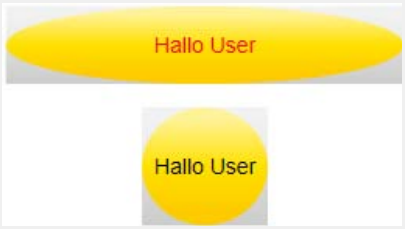
BEISPIEL 14.33: Verwendung der *Button.Background*-Eigenschaft

XAML

```
        <ControlTemplate TargetType="{x:Type Button}">
          <Grid HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
                ClipToBounds="False">
            <Rectangle Fill="{TemplateBinding Property=Background}"/>
            <Ellipse Name="elli">
```

Ergebnis

Starten Sie das Programm mit dieser Änderung, taucht im Hintergrund zunächst der Default-Farbverlauf eines Buttons auf (so ist *Background* für einen *Button* auch gesetzt).



Sie können aber über den Style gleich noch eine andere Hintergrundfarbe auswählen:

```
    <Setter Property="FontSize" Value="14" />
    <Setter Property="Background" Value="Blue" />
  </Style>
```

Werfen wir noch einen Blick auf das *ContentPresenter*-Element in unserem obigen Template. Dessen Content stammt vom ursprünglichen Element ab und bietet damit ebenfalls die Möglichkeit, komplexe Controls "zusammenzubasteln".

BEISPIEL 14.34: Wir definieren einen zusätzlichen Button mit einer Grafik, der Button übernimmt den vorliegenden Style.

XAML

```
  <Button Height="76" Margin="10" Width="113">
   <StackPanel Orientation="Horizontal">
     <Image Source="Images/flash.png" Width="26" Height="26" Margin="0,0,10,0"/>
     <TextBlock VerticalAlignment="Center">Action</TextBlock>
   </StackPanel>
  </Button>
```

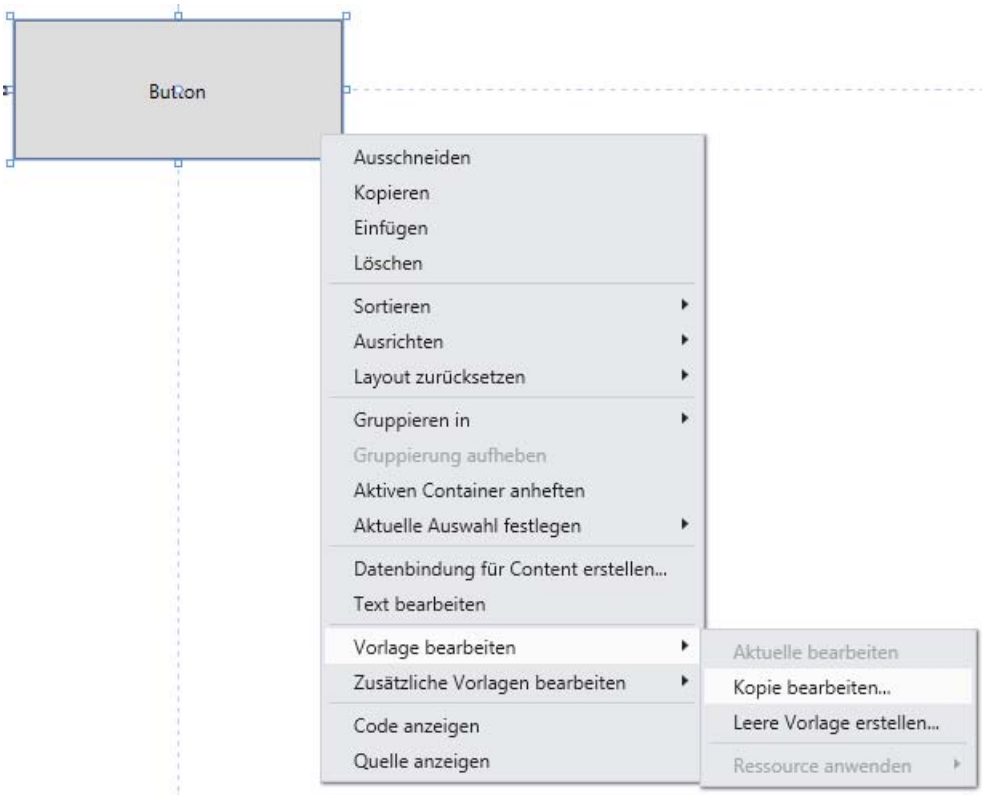
Ergebnis

Das zusammengewürfelte Endergebnis (Grafik und Text per *Content,* Grundlayout per *Template)* präsentiert sich nach wie vor als vollwertiger Button:

14.7.2 Template abrufen und verändern

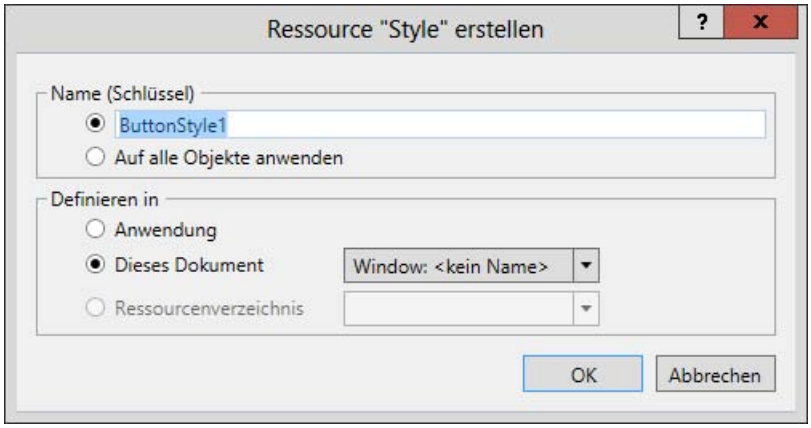
Möchten Sie von einem vorhanden Control das Template abrufen und eventuell verändern, ist dies seit der Einführung von Visual Studio 2012 kein Problem mehr[1].

Sie können von jedem Control direkt aus dem Designer heraus eine Kopie des Default-Templates editieren. Markieren Sie das betreffende Control im WPF-Editor-Fenster und klicken Sie auf die rechte Maustaste. Über das Kontextmenü rufen Sie *Vorlage bearbeiten|Kopie* bearbeiten auf:



Im folgenden Dialog bestimmen Sie, ob der Style per Key (benannt) oder per Typ zugeordnet werden soll und wo der Style erstellt wird (siehe folgende Abbildung). Auf das Speichern im aktuellen Window sollten Sie aus Gründen der Übersichtlichkeit verzichten.

[1] In früheren Versionen waren Sie auf die Anwendung *Expression Blend* angewiesen, mittlerweile ist der WPF-Editor in Visual Studio massiv aufgerüstet worden.



BEISPIEL 14.35: Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

XAML

```
<Application x:Class="WpfApplication2.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
```

Hier geht es mit der eigentlichen Definition des Buttons los, bei der zunächst einige Einstellungen zugewiesen werden:

```
        <Style TargetType="{x:Type Button}">
            <Setter Property="FocusVisualStyle">
                <Setter.Value>
                    <Style>
                        <Setter Property="Control.Template">
                            <Setter.Value>
                                <ControlTemplate>
                                    <Rectangle Margin="2" SnapsToDevicePixels="True"
                                        Stroke="{DynamicResource {x:Static
                                        SystemColors.ControlTextBrushKey}}"
                                        StrokeThickness="1" StrokeDashArray="1 2"/>
                                </ControlTemplate>
                            </Setter.Value>
                        </Setter>
                    </Style>
                </Setter.Value>
            </Setter>
            <Setter Property="Background" Value="#FFDDDDDD"/>
            <Setter Property="BorderBrush" Value="#FF707070"/>
```

Hier wird direkt eine Systemressource genutzt, die Anpassung erfolgt dynamisch, d.h. bei Änderungen in der Systemsteuerung:

```
            <Setter Property="Foreground"
```

BEISPIEL 14.35: Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

```
                Value="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"/>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
        <Setter Property="Template">
            <Setter.Value>
```

Das Default-Template als Kopie:

```
                <ControlTemplate TargetType="{x:Type Button}">
                    <Border x:Name="border" BorderBrush="{TemplateBinding BorderBrush}"
                        BorderThickness="{TemplateBinding BorderThickness}"
                        Background="{TemplateBinding Background}"
                        SnapsToDevicePixels="True">
                        <ContentPresenter x:Name="contentPresenter"
                          ContentTemplate="{TemplateBinding ContentTemplate}"
                          Content="{TemplateBinding Content}"
                          ContentStringFormat="{TemplateBinding ContentStringFormat}"
                          Focusable="False" HorizontalAlignment="{TemplateBinding
                          HorizontalContentAlignment}" Margin="{TemplateBinding Padding}"
                          RecognizesAccessKey="True"
                          SnapsToDevicePixels="{TemplateBinding SnapsToDevicePixels}"
                          VerticalAlignment="{TemplateBinding
                        VerticalContentAlignment}"/>
                    </Border>
```

Die Reaktion auf Eigenschaften-Trigger:

```
                    <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="True">
                            <Setter Property="BorderBrush" TargetName="border"
                    Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
```

Reaktion auf Mausbewegung:

```
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFBEE6FD"/>
                            <Setter Property="BorderBrush" TargetName="border"
                                    Value="#FF3C7FB1"/>
                        </Trigger>
```

Reaktion auf Klicken:

```
                        <Trigger Property="IsPressed" Value="True">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFC4E5F6"/>
                            <Setter Property="BorderBrush" TargetName="border"
```

BEISPIEL 14.35: Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

```
                                    Value="#FF2C628B"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="False">
                            <Setter Property="Background" TargetName="border"
                                    Value="#FFF4F4F4"/>
                            <Setter Property="BorderBrush" TargetName="border"
                                    Value="#FFADB2B5"/>
                            <Setter Property="TextElement.Foreground"
                                    TargetName="contentPresenter" Value="#FF838383"/>
                        </Trigger>
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

  </Application.Resources>
</Application>
```

HINWEIS: Ändern Sie in diesem Template beispielsweise die Farben im *IsMouseOver*-Trigger (siehe fett hervorgehobene Zeilen), wird sich das Aussehen alle Schaltflächen in Ihrer Anwendung ändern.

Hauptzielgruppe dieses Features dürfte jedoch nicht der Programmierer sondern der Designer der Anwendung sein. Gleiches trifft ebenfalls auf das im folgenden Abschnitt behandelte Storyboard zu.

14.8 Transformationen, Animationen, StoryBoards

Im Folgenden wollen wir in einem "Schnelldurchlauf für Programmierer" noch einen Blick auf WPF-typische Features werfen, auch wenn diese im Allgemeinen nicht zum Hautarbeitsgebiet des Programmierers gehören.

14.8.1 Transformationen

Mit Hilfe von Transformationen können Sie in WPF das optische Standardverhalten problemlos verändern, ohne sich um Templates oder Styles kümmern zu müssen. Sie können die Größe, Position, Drehung und Verzerrung der betroffenen Controls über die einfache Zuweisung einer entsprechenden Transformation ändern (statische Änderung).

HINWEIS: Damit sind diese Operationen auch die Vorstufe für einfache Animationen (dynamische Änderungen in Abhängigkeit von der Zeit).

Folgende Möglichkeiten stehen Ihnen zur Verfügung:

Transformation	Beschreibung
RotateTransform	Element um einen bestimmten Winkel drehen
ScaleTransform	Element vergrößern/verkleinern
SkewTransform	Element verformen
TranslateTransform	Element verschieben
MatrixTransform	Zusammenfassen der obigen Transformationen per 3x3 Transformationsmatrix

Lassen Sie uns nun an einfachen Beispielen die Wirkung der jeweiligen Transformation demonstrieren.

Drehen mit RotateTransform

Mit *RotateTransform* realisieren Sie eine Drehung des Elements im Uhrzeigersinn. Den Drehpunkt können Sie optional festlegen, per Default ist dies die linke obere Ecke.

BEISPIEL 14.36: Button um 40° drehen

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <RotateTransform Angle="40"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

Das *Angle*-Attribut bestimmt den Drehwinkel, um den per *CenterX*- und *CenterY*-Attribut festgelegten Drehpunkt. Natürlich können Sie für eine andere Drehrichtung auch negative Werte übergeben.

HINWEIS: Durch die Rotation wird das Koordinatensystem des gedrehten Elements verändert!

Skalieren mit ScaleTransform

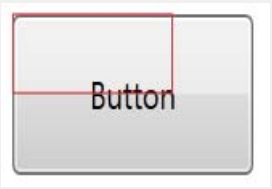
Soll ein Element skaliert werden, nutzen Sie eine *ScaleTransform*.

BEISPIEL 14.37: Button skalieren

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <ScaleTransform ScaleX="1.5" ScaleY="2"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Mit den *ScaleX*- und *ScaleY*-Attributen bestimmen Sie den Skalierungsfaktor für die X- bzw. Y-Achse. Mit *CenterX*- und *CenterY* bestimmen Sie den Fixpunkt von dem aus die Skalierung gestartet wird, dies ist per Default die linke obere Ecke.

Verformen mit SkewTransform

Mit *SkewTransform* verformen Sie das Koordinatensystem um bestimmte Winkel.

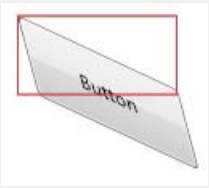
BEISPIEL 14.38: Verformen des Koordinatensystems

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <SkewTransform AngleY="25" AngleX="15" />
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

BEISPIEL 14.38: Verformen des Koordinatensystems

Ergebnis



Den Verformungsgrad bestimmen Sie mit den Attributen *AngleX*- und *AngleY*. *CenterX* und *CenterY* legen den Ursprungspunkt fest.

Verschieben mit TranslateTransform

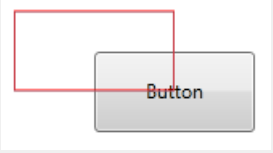
Auch die Verschiebung eines Elements ist mit *TranslateTransform* kein Problem. Sie können mit den X- und Y-Attributen die horizontale bzw. vertikale Verschiebung bestimmen.

BEISPIEL 14.39: Button verschieben

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <TranslateTransform X="50" Y="25"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis



Und alles zusammen mit TransformGroup

Sicher sind Sie auch schon versucht gewesen, mehrere der obigen Effekte gleichzeitig zu realisieren. Allerdings dürfte Ihnen die Syntax-Prüfung des XAML-Editors hier einen Strich durch die Rechnung gemacht haben.

HINWEIS: Um mehrere Transformationen gleichzeitig zuzuweisen, müssen Sie eine *TransformGroup* verwenden.

BEISPIEL 14.40: Mehrere Transformationen gleichzeitig anwenden

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
        <Button.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
                <RotateTransform Angle="45"></RotateTransform>
                <TranslateTransform X="50" Y="25"/>
            </TransformGroup>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

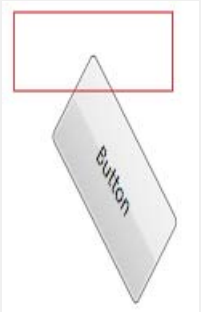
Doch Achtung: Hier spielt die Reihenfolge in der Gruppe eine bedeutende Rolle, wie folgende kleine Änderung (erst Drehung, dann Skalierung) zeigt. Ursache ist die Veränderung des Koordinatensystems des betroffenen Controls.

BEISPIEL 14.41: Veränderung der Transformationsreihenfolge

XAML

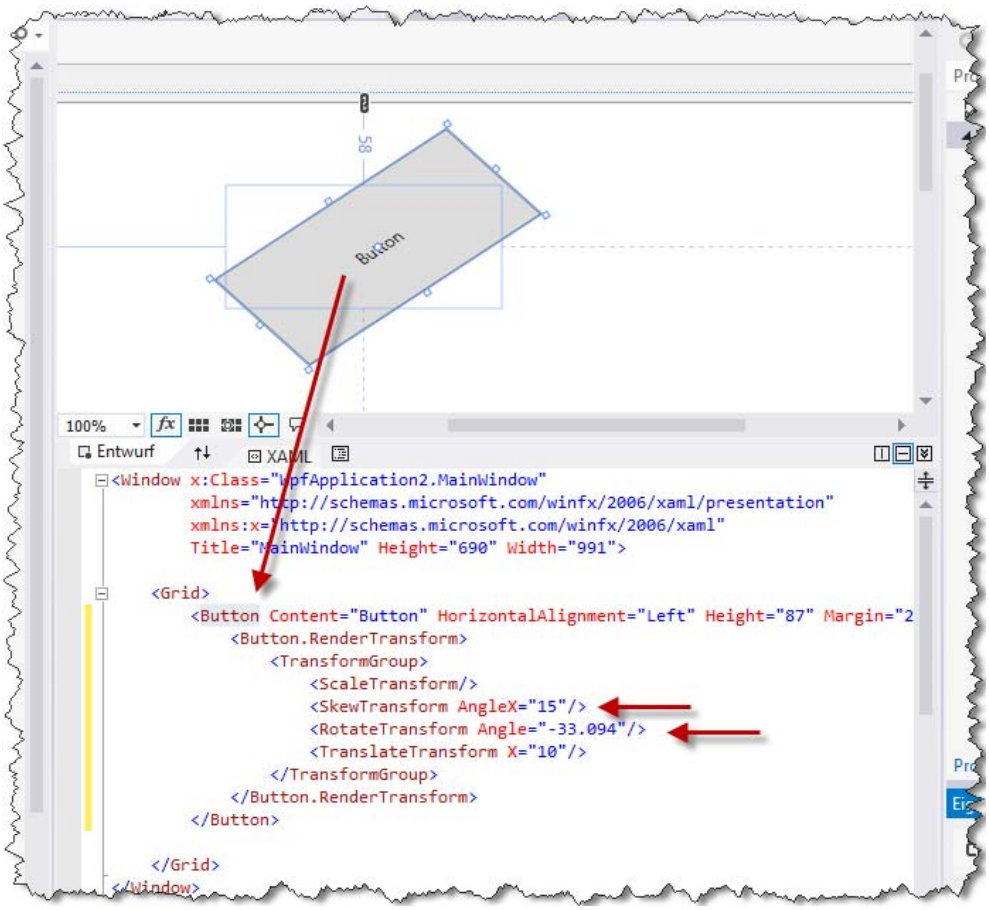
```
            <TransformGroup>
                <RotateTransform Angle="45"></RotateTransform>
                <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
                <TranslateTransform X="50" Y="25"/>
            </TransformGroup>
```

Ergebnis

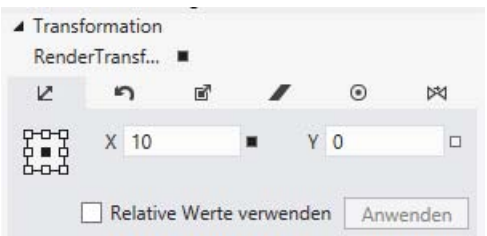


Hilfe durch den WPF-Editor

Wem die obigen Ausführungen zu kompliziert und wenig intuitiv waren, der kann es auch einfacher haben. Nutzen Sie einfach den in Visual Studio integrierten WPF-Editor, dieser ist seit der Version 2012 sogar ganz brauchbar und bietet unter anderem auch die Möglichkeit, die RotateTransformation per Maus umzusetzen.



Im Eigenschaftenfenster können Sie zusätzlich die anderen Transformationsarten getrennt parametrieren:



14.8.2 Animationen mit dem StoryBoard realisieren

Für die Realisierung von Animationen werden in WPF so genannte Storyboards verwendet, die wiederum einzelne oder mehrere Animationen (zeitliche Veränderungen von Eigenschaften) enthalten können. Storyboards können wiederum über bestimmte Ereignis-Trigger ausgelöst, angehalten, fortgesetzt oder auch beendet werden (alternativ natürlich auch per Code).

Ein erstes einfaches Beispiel soll die prinzipielle Vorgehensweise beim Animieren einer Eigenschaft (in diesem Fall der Transparenz) demonstrieren.

BEISPIEL 14.42: Ausblenden eines Buttons, wenn die Maus darüber bewegt wird

XAML

```
<Canvas>
```

Zunächst den Button definieren:

```
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
            Name="button1" Width="100" >
```

Wir reagieren mit einem *Trigger* auf das Hineinbewegen der Maus:

```
        <Button.Triggers>
            <EventTrigger RoutedEvent="Button.MouseEnter">
```

Auf Grund des ausgelösten Trigger-Ereignisses wird das folgende *StoryBoard* ausgeführt:

```
                <BeginStoryboard>
                    <Storyboard x:Name="Storyboard1">
```

Das *StoryBoard* enthält eine *DoubleAnimation* (Verändern einer *Double*-Eigenschaft) mit einer Zeitdauer (*Duration*) von 4 Sekunden:

```
                        <DoubleAnimation Duration="0:0:4"
```

Ziel der Eigenschaftsänderung ist *button1*:

```
                         Storyboard.TargetName="button1"
```

Die zu ändernde Eigenschaft ist *Opacity*:

```
                         Storyboard.TargetProperty="Opacity"
```

Die Eigenschaft wird in der o.g. Zeitdauer von 1 auf 0 geändert:

```
                         From="1" To="0" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Button.Triggers>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50" Name="rectangle1"
               Stroke="Red" Width="100" />
</Canvas>
```

Das war hoffentlich nicht allzu abschreckend, es geht teilweise auch einfacher und alternativ könnten Sie auch mit sinnvollen Werkzeugen den obigen XAML-Code erstellen.

Animation per C#-Code realisieren

Im obigen Fall müssen Sie immer ein StoryBoard einsetzen, um die Animation(en) zu kapseln. Etwas einfacher geht es, wenn Sie lediglich eine Animation per C#-Code realisieren wollen. In diesem Fall erstellen Sie einfach eine Instanz der gewünschten Animation (davon gibt es je nach Zieleigenschaft unterschiedliche), parametrieren diese und starten die Animation, indem Sie diese an die *BeginAnimation*-Methode des gewünschten Controls übergeben.

BEISPIEL 14.43: Eine einfache Animation per Code definieren und ausführen

C#

```
...
using System.Windows.Media.Animation;
...
        private void button2_Click(object sender, RoutedEventArgs e)
        {
```

Instanz erstellen:

```
            DoubleAnimation ani = new DoubleAnimation();
```

Parametrieren:

```
            ani.From = 1;
            ani.To = 0;
```

Starten (Transparenz ändern):

```
            button2.BeginAnimation(Button.OpacityProperty, ani);
        }
```

Das war doch gar nicht so schwierig, oder?

Animation per Code steuern

Vielleicht dämmert es Ihnen schon, komplexe Animationen bzw. die Zusammenfassung mehrerer Animationen als Storyboard sind kaum für die tägliche Praxis des C#-Programmierers geeignet. Abgesehen davon, dass Sie Unmengen von C#-Code erzeugen, fehlt bei vielen Animationen einfach die Vorstellungskraft. Dauernde Programmstarts zum Ausprobieren der Effekte zehren auch an den Nerven und kosten Zeit. Ganz nebenbei ist auch die Parametrierung vieler Eigenschaften mit C# eine Pein, hier kann XAML seine Vorteile deutlich ausspielen.

Viel besser ist es, die Storyboards mit einem Programm wie Microsoft Blend zu erstellen und nachträglich in Ihre Anwendung einzufügen. Zum Starten der Animation können Sie entweder, wie bereits gezeigt, einen Trigger verwenden, oder Sie nutzen das *Storyboard* per Code. Dazu stellt die *Storyboard*-Klasse mehrere Methoden bereit, mit denen Sie die Animation gezielt kontrollieren können:

Methode	Beschreibung
Begin	Animationen des *Storyboards* starten.
Pause	Wiedergabe anhalten.
Resume	Wiedergabe fortsetzen.
Seek	Bei der Wiedergabe zu einer Position im *Storyboard* springen. Verwenden Sie eine *TimeSpan-Wert*.
Stop	Wiedergabe anhalten und Wiedergabeposition zurück setzen.

HINWEIS: Auf das Ende der Animationen im *Storyboard* können Sie mit dessen *Completed*-Ereignis reagieren.

BEISPIEL 14.44: Als Ressource definierte Animation per Code starten

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
        Title="MainWindow" Height="350" Width="525">
```

Als Window-Ressource definieren wir ein *Storyboard*:

```
    <Window.Resources>
```

Achten Sie darauf, einen *Key* zu vergeben:

```
        <Storyboard x:Key="storyboard2">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button1"
                             Storyboard.TargetProperty="Width"  To="300" />
        </Storyboard>
    </Window.Resources>
        <Canvas>
        <Button Canvas.Left="100" Canvas.Top="100" Content="Button" Height="50"
                Name="button1" Width="100" Click="button1_Click">
        </Button>
...
```

C#

Zunächst den Namespace importieren:

```
...
using System.Windows.Media.Animation;
...
```

Zur Laufzeit können wir unser *Storyboard* suchen und mit der *Begin*-Methode starten:

```
       private void button1_Click(object sender, RoutedEventArgs e)
        {
            Storyboard sb = FindResource("storyboard2") as Storyboard;
            if (sb != null) sb.Begin();
```

Ist für das *Storyboard* kein *TargetName* vorgegeben, können Sie das *Storyboard* auch auf jedes andere Control anwenden, wenn Sie dieses an die *Begin*-Methode übergeben:

BEISPIEL 14.44: Als Ressource definierte Animation per Code starten

```
            if (sb != null) sb.Begin(button2);
        }
```

HINWEIS: Ein Klick auf eine andere Taste könnte beispielsweise mit *Storyboard.Stop* die Animation anhalten.

Selbstverständlich können Sie ein in den Ressourcen abgelegtes Storyboard auch per XAML einbinden bzw. starten. In diesem Fall benötigen Sie nicht eine Zeile C#-Code, können aber die Animationen (bzw. die übergeordneten Storyboards) zentral verwalten.

BEISPIEL 14.45: Alternative zum vorhergehenden Beispiel

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
    <Window.Resources>
        <Storyboard x:Key="storyboard2">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button1"
                             Storyboard.TargetProperty="Width" To="300" />
        </Storyboard>
    </Window.Resources>
...
        <Button Canvas.Left="50" Canvas.Top="206" Content="Button" Height="23"
                Name="button4" Width="75" >
```

Per Trigger starten wir ein *Storyboard*:

```
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.MouseEnter">
```

Hier weisen wir die Ressource zu:

```
                    <BeginStoryboard Storyboard="{StaticResource storyboard2}" />
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </Canvas>
</Window>
```

Haben Sie ein *Storyboard* ohne *TargetName* (universelle Verwendung), müssen Sie diesen beim Einbinden der Ressource angeben, um auch das Ziel der Animation zu bestimmen:

BEISPIEL 14.46: Ziel bestimmen

XAML

```
...
    <EventTrigger RoutedEvent="Button.MouseEnter">
        <BeginStoryboard Storyboard="{StaticResource storyboard4}"
                         Storyboard.TargetName="button4" />
```

BEISPIEL 14.46: Ziel bestimmen

```
    </EventTrigger>
...
```

Mehrere Animationen zusammenfassen

Dass eine Animation nur auf der linearen Änderung einer Eigenschaft basiert, dürfte wohl selten der Fall sein. Meist werden mehrere Eigenschaften gleichzeitig geändert. Auch das ist mit dem *Storyboard* kein Problem, wie es das folgende Beispiel zeigt:

BEISPIEL 14.47: *Storyboard* mit drei Animationen

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Storyboard x:Key="storyboard3">
            <DoubleAnimation Duration="0:0:4" Storyboard.TargetProperty="Width" To="300" />
            <DoubleAnimation Duration="0:0:4"
                     Storyboard.TargetProperty="RenderTransform.Angle" To="360" />
            <ColorAnimation  Duration="0:0:3"
                     Storyboard.TargetProperty="Foreground.Color" From="Red"  To="Blue" />
        </Storyboard>
    </Window.Resources>
```

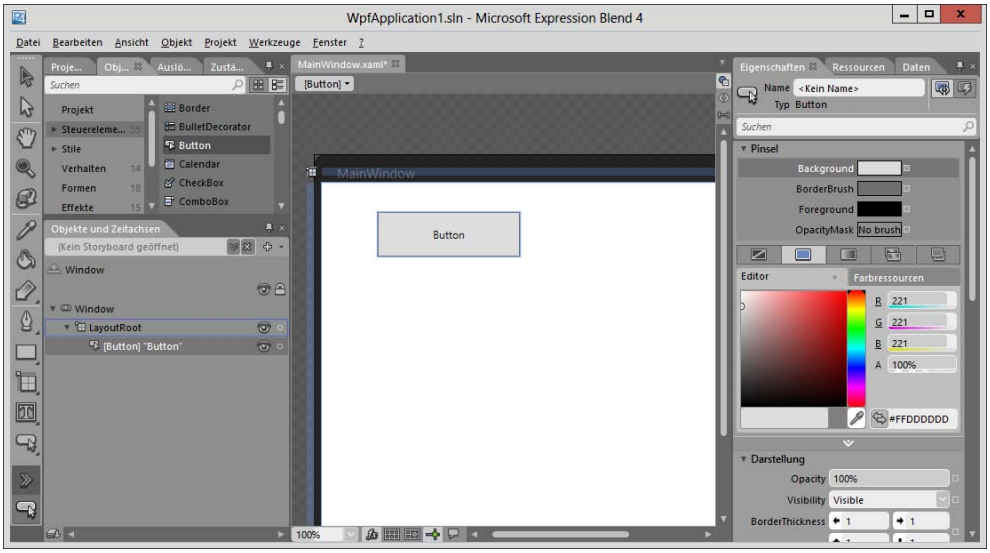
Das soll zu diesem Thema genügen. Auch wenn WPF im Bereich "Animationen" fast unbegrenzte Möglichkeiten bietet, so sprechen diese doch kaum den Programmierer sondern eher den Designer der Anwendung an. Machen Sie sich also nicht die Mühe, komplexe Storyboards per XAML oder gar C#-Quellcode zu erstellen, sondern nutzen Sie hier die Vorteile von Microsoft Expression Blend, erstellen Sie damit interaktiv den entsprechende XAML-Code und fügen Sie diesen in die Ressourcen Ihrer Anwendung ein. Damit ersparen Sie sich viele graue Haare und haben mehr Zeit für die eigentliche Anwendungsentwicklung.

14.9 Praxisbeispiel

14.9.1 Arbeiten mit Microsoft Expression Blend

Seit der Einführung von WPF bietet Microsoft neben Visual Studio auch eine zweite Entwicklungsumgebung an, die sich jedoch gezielt an den Designer wendet. Die Rede ist von *Expression Blend*, das in der aktuellen Version 4 vorliegt.

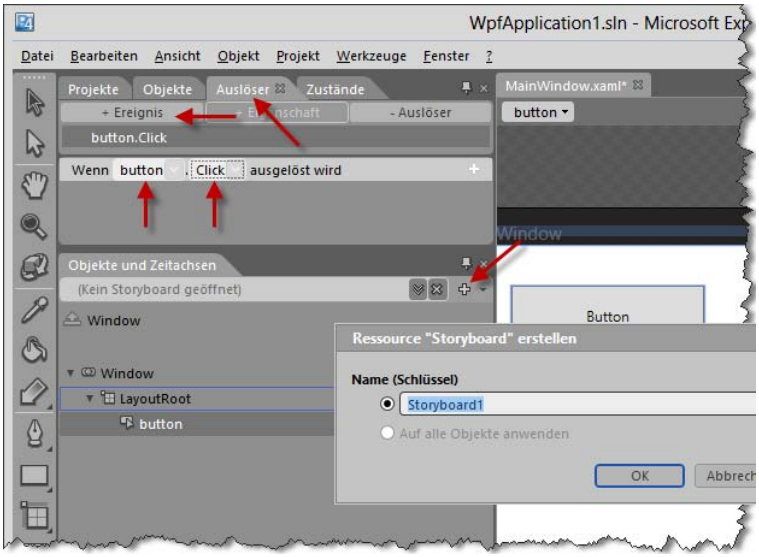
Für den Visual Studio-Entwickler dürfte die Oberfläche von Expression Blend auf den ersten Blick etwas gewöhnungsbedürftig sein, was sicher auch an der "eigenartigen" Farbgebung liegt:



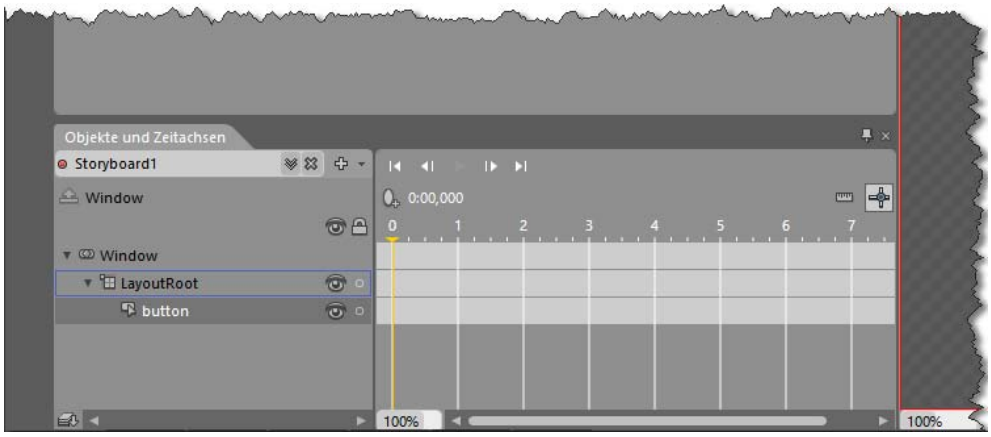
Der wesentlichste Unterschied zu Visual Studio zeigt sich zunächst im Grundansatz, dass die Entwicklung von Aktionen (Trigger) quasi per Assistent und nicht per Quellcode abläuft. C#-Quelltext wird von Expression Blend stiefmütterlich behandelt, der Designer soll mit den vorhandenen Triggern etc. arbeiten.

Eine Animation realisieren

Mit dem Klick auf eine Schaltfläche soll eine Animation gestartet werden. Dazu wählen Sie zunächst im "Triggers"-Bereich die Taste "+Event" und bestimmen nachfolgend über die beiden Auswahllisten welches Objekt (Button) und welche Aktion (Click) zugeordnet werden:



Nachfolgend steht die Frage, ob Sie eine neue Timeline (Storyboard Ressource) erstellen wollen. Diese Timeline dürfte Hobbyfilmern sicher bekannt vorkommen, handelt es sich doch um eine Zeitachse, bei der Sie Objekt-Eigenschaften zu bestimmten Zeitpunkten definiert setzen können:



Die Verwendung ist recht simpel, wählen Sie beispielsweise in der Timeline den Zeitpunkt 5 (Sekunden) und verschieben Sie jetzt ein Control und ändern Sie dessen Größe[1].

Der XAML-Code

Nach dem Umschalten in die XAML-Ansicht können Sie bereits den erzeugten Code (Storyboard) einsehen:

```
...
 <Window.Resources>
   <Storyboard x:Key="Storyboard1">
```

Eine Transformation (Verschiebung X-Achse):

```
     <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.TargetName="button"
      Storyboard.TargetProperty="(UIElement.RenderTransform).(TransformGroup.Children)[3].
      (TranslateTransform.X)">
       <SplineDoubleKeyFrame KeyTime="00:00:05" Value="96.5"/>
     </DoubleAnimationUsingKeyFrames>
```

Eine Transformation (Verschiebung Y-Achse):

```
     <DoubleAnimationUsingKeyFrames BeginTime="00:00:00" Storyboard.TargetName="button"
        Storyboard.TargetProperty="(UIElement.RenderTransform).(TransformGroup.Children)[3].
        (TranslateTransform.Y)">
       <SplineDoubleKeyFrame KeyTime="00:00:05" Value="97.5"/>
     </DoubleAnimationUsingKeyFrames>
...
   </Storyboard>
```

[1] Im Kopf der Design-Ansicht sollte "Zeitachse-Aufzeichnung ist an" angezeigt werden.

```
</Window.Resources>
```

Und hier werden Storyboard und Ereignis miteinander verknüpft:

```
<Window.Triggers>
  <EventTrigger RoutedEvent="ButtonBase.Click" SourceName="button">
    <BeginStoryboard x:Name="OnClick2_BeginStoryboard"
                     Storyboard="{StaticResource OnClick1}"/>
  </EventTrigger>
</Window.Triggers>
```

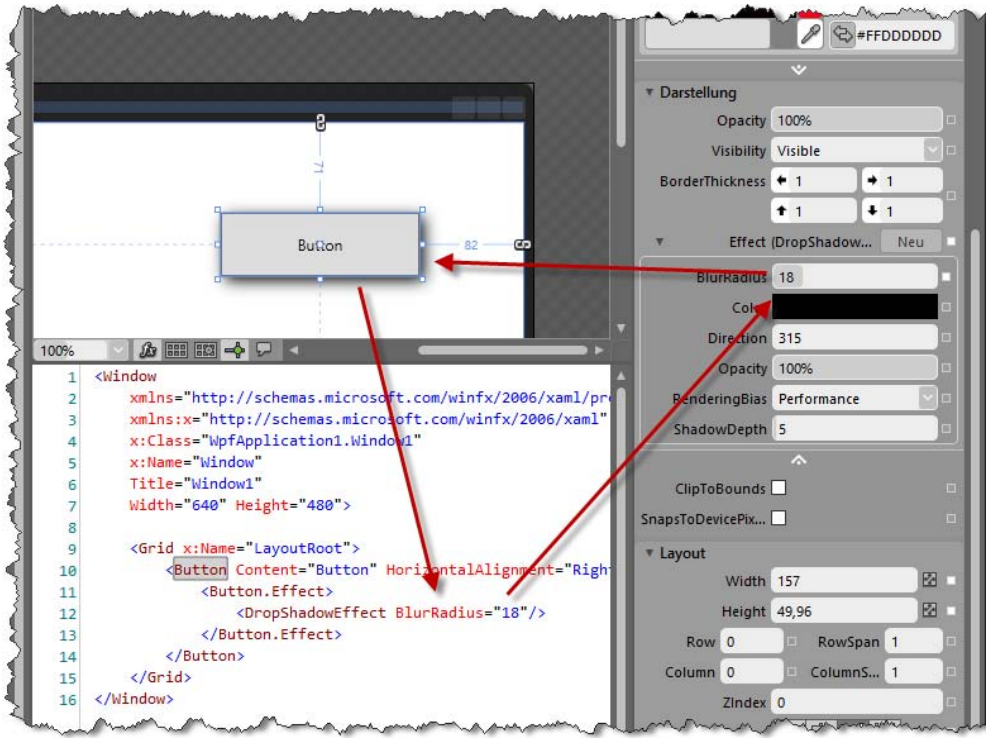
C# oder VB-Code für die Ereignisbehandlung sind an dieser Stelle nicht erzeugt worden und auch nicht notwendig.

Test

Starten Sie das Projekt mit F5 und klicken Sie auf die Schaltfläche. Diese sollte sich jetzt mit einer Animation bewegen und gleichzeitig drehen. Sollen diese Aktionen getrennt ablaufen, müssen Sie in der Timeline mehrere Zwischenpunkte setzen und die jeweiligen Aktionen ausführen.

Bemerkung

Auch die Parametrierung von Effekten über das Eigenschaftenfenster oder den XAML-Quellcode geht dank interaktiver Anzeige schnell von der Hand.



Doch was nützen Ihnen als C#-Programmierer all diese Möglichkeiten?

Zwei Szenarien sind denkbar:

- Sie erstellen die WPF-Anwendung komplett selbst und nutzen die erweiterten Möglichkeiten von Expression Blend, um Storyboards und komplexe Oberflächendefinitionen zu erstellen. Die erzeugten XAML-Daten können Sie in Ihr Visual Studio-Projekt (z.B. per Zwischenablage) importieren, oder Sie öffnen das betreffende Expression Blend-Projekt gleich mit Visual Studio 2012.
- Sie arbeiten mit einem Designer zusammen, der das Projekt in Expression Blend optisch aufbereitet und Sie können den erforderlichen Quellcode beitragen. Ein Doppelklick auf die dem Window zugeordnete C#-Datei bewirkt, dass diese gleich in Visual Studio geöffnet wird, alternativ öffnen Sie das Projekt komplett mit Visual Studio.

Kapitel 15

WPF-Datenbindung

Nachdem wir schon an der einen oder anderen Stelle auf Datenbindung zurückgegriffen haben, wollen wir uns jetzt direkt mit dieser Thematik beschäftigen.

Im Unterschied zu den Windows Forms-Anwendungen sind Sie bei der Datenbindung nicht auf spezielle Controls angewiesen, in einer WPF-Anwendung kann fast jede Eigenschaft (Abhängigkeitseigenschaft) an andere Eigenschaften gebunden werden.

Als Datenquelle können Sie beispielsweise:

- Eigenschaften anderer WPF-Controls (Elemente),
- Ressourcen,
- XML-Elemente oder
- beliebige Objekte (auch ADO.NET-Objekte, z.B. *DataTable)*

verwenden.

HINWEIS: Im Rahmen dieses Buchs werden wir uns auf einen ersten Überblick beschränken, zu den Details und Hintergründen verweisen wir auf unser Buch [Datenbankprogrammierung mit Visual C# 2012].

15.1 Grundprinzip

Zunächst wollen wir Ihnen das Grundprinzip der Datenbindung in WPF an einem recht einfachen Beispiel demonstrieren.

BEISPIEL 15.1: Datenbindung zwischen *Slider* und *ProgressBar*

XAML

Fügen Sie in ein *Window* einen *ProgressBar* und einen *Slider* ein. Mit dem *Slider* soll der aktuelle Wert des *ProgressBar* direkt und ohne zusätzlichen Quellcode verändert werden.

```
<StackPanel>
```

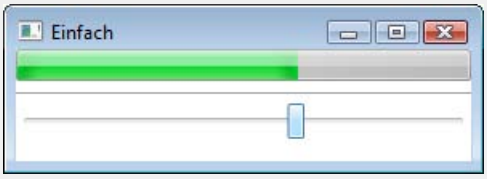
BEISPIEL 15.1: Datenbindung zwischen *Slider* und *ProgressBar*

Hier sehen Sie auch schon den Ablauf: Das Ziel (*ProgressBar*) bindet seine Eigenschaft *Value* an die Quelle (*Slider*) mit deren Eigenschaft *Value*.

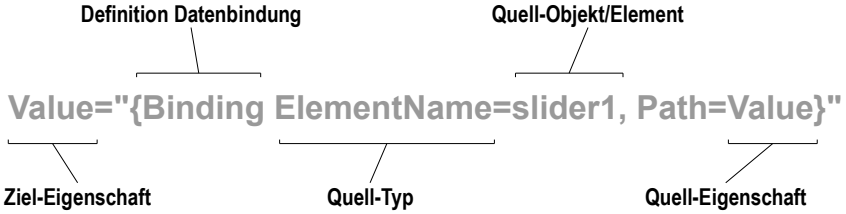
```
  <ProgressBar Height="20" Name="progressBar1" Maximum="100"
               Value="{Binding ElementName=slider1, Path=Value}"/>
  <Separator Height="10"/>
  <Slider Name="slider1" Maximum="100" />
</StackPanel>
```

Ergebnis

Zur Laufzeit können Sie den *Slider* beliebig verändern, der *ProgressBar* passt sofort seinen Wert an:



Sehen wir uns noch einmal die Syntax im Detail an:



HINWEIS: Kann die Quelleigenschaft nicht automatisch in den Datentyp der Zieleigenschaft konvertiert werden, können Sie zusätzlich einen Typkonverter angeben (siehe dazu Seite 820).

15.1.1 Bindungsarten

Das vorhergehende Beispiel zeigte bereits recht eindrucksvoll, wie einfach sich Eigenschaften verschiedener Objekte miteinander verknüpfen lassen. Doch das ist noch nicht alles. Über ein zusätzliches Attribut *Mode* lässt sich auch bestimmen, in welche Richtungen die Bindung aktiv ist, d h., ob die Werte nur von der Quelle zum Ziel oder auch umgekehrt übertragen werden. Die folgende Tabelle zeigt die möglichen Varianten:

Typ	Beschreibung
OneTime	Mit der Initialisierung wird der Wert einmalig von der Quelle zum Ziel kopiert. Danach wird die Bindung aufgehoben.
OneWay	Der Wert wird nur von der Quelle zum Ziel übertragen (readonly). Ändert sich der Wert des Ziels, wird die Bindung aufgehoben.
OneWayToSource	Der Wert wird vom Ziel zur Quelle übertragen (writeonly). Ändert sich der Wert der Quelle, bleibt die Bindung erhalten, eine Wertübertragung findet jedoch nicht statt.
TwoWay	(meist Defaultwert[1]) Werte werden zwischen Quelle und Ziel in beiden Richtungen übertragen.

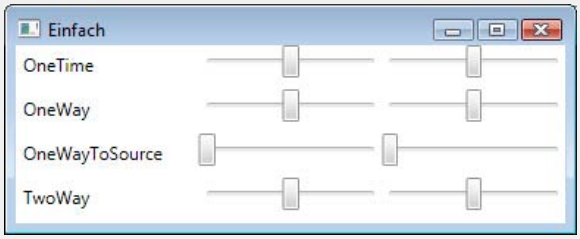
BEISPIEL 15.2: Testen der verschiedenen Bindungsarten

XAML

```
...
    <StackPanel Grid.Column="2">
      <Slider Name="sl2" Maximum="100" Height="30"
              Value="{Binding ElementName=sl1, Path=Value, Mode=OneTime}"/>
      <Slider Name="sl4" Maximum="100" Height="30"
              Value="{Binding ElementName=sl3, Path=Value, Mode=OneWay}"/>
      <Slider Name="sl6" Maximum="100" Height="30"
              Value="{Binding ElementName=sl5, Path=Value, Mode=OneWayToSource}"/>
      <Slider Name="sl8" Maximum="100" Height="30"
              Value="{Binding ElementName=sl7, Path=Value, Mode=TwoWay}"/>
    </StackPanel>
...
```

Ergebnis

Verschieben Sie ruhig einmal die *Slider* im Testprogramm. Jeweils der linke und der rechte *Slider* bilden eine Datenbindung und sollten auch das entsprechende Verhalten zeigen:



15.1.2 Wann eigentlich wird die Quelle aktualisiert?

Im obigen Beispiel scheint alles ganz einfach zu sein, Sie ziehen an einem Schieberegler und der andere bewegt sich mit. Doch was ist, wenn Sie beispielsweise eine *TextBox* in einer Datenbindung verwenden? Hier steht die Frage, **wann** der "gewünschte" Wert wirklich in der *TextBox* steht.

[1] Bei Bindung an eine *ItemsSource* wird per Default OneWay -Binding verwendet.

Eine eingegebene Ziffer ist vielleicht nicht der richtige Wert, kann aber schon als gültiger Inhalt interpretiert werden. Nicht in jedem Fall möchte man deshalb sofort einen Datenaustausch zwischen Ziel und Quelle zulassen (bei *TwoWay* oder *OneWayToSource*).

Über das optionale Attribut *UpdateSourceTrigger* haben Sie direkten Einfluss darauf, wann die Aktualisierung **der Quelle** durchgeführt wird. Vier Varianten bieten sich dabei an:

- *Default*
  Meist wird das *PropertyChanged*-Ereignis für die Datenübernahme genutzt, bei einigen Controls kann es auch *LostFocus* sein.
- *Explicit*
  Die Datenübernahme muss "manuell" per *UpdateSource*-Methode ausgelöst werden.
- *LostFocus*
  Die Datenübernahme erfolgt bei Fokusverlust des Ziels.
- *PropertyChanged*
  Die Datenübernahme erfolgt mit jeder Werteänderung. Dies kann bei komplexeren Abläufen zu Problemen führen, da der Abgleich, z.B. bei einem Schieberegler/Scrollbar, recht häufig vorgenommen wird.

BEISPIEL 15.3: Explizite Datenübernahme nur per Enter-Taste

XAML

```
<StackPanel>
  <TextBox Name="txt1">Hallo</TextBox>
  <TextBox Name="txt2"
      Text="{Binding ElementName=txt1, Path=Text, UpdateSourceTrigger=Explicit}"
      KeyDown="TextBox_KeyDown"/>
</StackPanel>
```

C#

```
    private void TextBox_KeyDown(object sender, KeyEventArgs e)
     {
         if (e.Key== Key.Enter)
             txt2.GetBindingExpression(TextBox.TextProperty).UpdateSource();
     }
```

HINWEIS: Da die Bindung im XAML-Code vorgenommen wurde, müssen wir im C#-Code erst mit *GetBindingExpression* das *BindingExpression*-Objekt abrufen, um die *UpdateSource*-Methode aufzurufen.

15.1.3 Geht es auch etwas langsamer?

Am obigen Beispiel konnten Sie es ja schon beobachten: Sie verschieben den *Slider* und der zweite *Slider* reagiert sofort. Soweit so schön. Was aber, wenn Sie erst nach einiger Zeit auf die Veränderung reagieren wollen?

Hier hilft die mit WPF 4.5 eingeführte Eigenschaft *Delay* weiter. Diese verzögert die Datenübergabe um den angegebenen Wert (Millisekunden). Das heißt, erst wenn die Zeit nach einer Änderung verstrichen ist, wird der Wert weitergegeben, jede Änderung in dieser Zeitspanne setzt den internen Timer zurück und lässt die Zeit erneut laufen. Bewegen Sie also den *Silder* dauernd hin und her passiert nichts, erst nach der letzten Bewegung und dem Ablaufen der Zeit wird auch die Änderung berücksichtigt.

BEISPIEL 15.4: Zeitverzögerung bei Datenbindung

XAML
```
...
    <Slider Name="sl0" Maximum="100" Height="30" Value="{Binding ElementName=sl9,
            Path=Value, Mode=TwoWay, Delay=500 }"/>
...
```

Was dem Einen oder Anderen als Spielerei vorkommen mag, ist ein fast unverzichtbares Feature im Zusammenhang mit größeren Datenmengen oder langsamen Datenverbindungen. Folgende Szenarien sind denkbar:

- 1 n-Beziehung,
  Änderungen in einer *ListBox* sollen sich nicht sofort auf die Detaildaten auswirken, sondern erst nachdem sich der Anwender für einen Datensatz final entschieden hat (Scrollen per Tastatur durch die Liste). Andernfalls kann es schnell zum Ruckeln oder Springen zwischen den Datensätzen kommen.
- Texteingaben,
  Nutzen Sie laufende Eingaben als Filter oder Suchwert, kann gerade bei großen Ergebnismengen eine Verzögerung bei der Eingabe auftreten (ein kurzer Filter mit Platzhalter hat meist große Ergebnismengen zur Folge).
- Anzeige großer Datenmengen,
  Mit der Auswahl in einer Liste soll eine größere Grafik angezeigt werden. Jede Änderung, z.B. beim Scrollen, führt im Normalfall zum Laden der Grafik. Hier ist eine entsprechende Verzögerung sinnvoll.

Alle obigen Fälle lassen sich natürlich auch mit einem eigenen *Timer* realisieren, aber warum kompliziert, wenn es jetzt auch wesentlich einfacher geht?

Eine Einschränkung sollten Sie allerdings beachten, auch wenn sie meist nicht von Bedeutung ist:

HINWEIS: Die Verzögerung gilt nur für eine Richtung der Datenbindung, d h. nur für das Control, dem auch die Verzögerung zugeordnet ist.

15.1.4 Bindung zur Laufzeit realisieren

Nicht immer werden Sie mit den schon zur Entwurfszeit definierten Datenbindungen auskommen. Es ist aber auch kein Problem, die Datenbindung erst zur Laufzeit per C#-Code zu realisieren. Alles was Sie dazu benötigen ist ein *Binding*-Objekt, dessen Konstruktor Sie bereits den *BindingPath* zuweisen können. Legen Sie anschließend noch die *BindingSource* sowie gegebenenfalls den

Mode (z.B. *OneWay*) fest. Letzter Schritt ist das eigentliche Binden mit der *SetBinding*-Methode des jeweiligen Controls.

BEISPIEL 15.5: Bindung zur Laufzeit realisieren

XAML

Unsere Testoberfläche:

```
<Window x:Class="Datenbindung.Bindung_Laufzeit"
...
        Title="Bindung_Laufzeit" Height="300" Width="300" Loaded="Window_Loaded">
    <StackPanel>
        <Label Name="Label1"></Label>
        <Button Name="Button1" Click="Button_Click">Test</Button>
    </StackPanel>
</Window>
```

C#

Mit dem Laden des Fensters erzeugen wir die Bindung wie oben beschrieben:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Wir binden an einen Button:

```
            Binding binding = new Binding("Content");
            binding.Source = Button1;
            binding.Mode = BindingMode.OneWay;
```

Binden an den Content:

```
            Label1.SetBinding(Label.ContentProperty, binding);
        }
```

Und hier verändern wir die Beschriftung des Buttons:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Button1.Content = "Ein neuer Text";
        }
    }
```

Test

Nach dem Start dürfte im Label zunächst "Test" stehen, die ursprüngliche Button-Beschriftung. Nach einem Klick auf die Schaltfläche ändern sich sowohl die Button-Beschriftung als auch die Label-Beschriftung.

Die Bindung selbst können Sie recht einfach wieder aufheben, indem Sie der Ziel-Eigenschaft der Bindung einen neuen Wert zuweisen.

BEISPIEL 15.6: Bindung zur Laufzeit aufheben

C#

Entweder so:

```
    Label1.Content = "Bindung beendet";
```

BEISPIEL 15.6: Bindung zur Laufzeit aufheben

Oder so:

```
Label1.ClearValue(Label.ContentProperty);
```

15.2 Binden an Objekte

Nachdem wir uns bereits mit dem Binden an Oberflächen-Elemente vertraut gemacht haben, wollen wir jetzt den Schritt hin zu selbstdefinierten Objekten gehen.

Prinzipiell bieten sich zwei Varianten der Instanziierung von Objekten an:

- Sie instanziieren die Objekte in XAML (in einem Ressource-Abschnitt).
- Sie instanziieren wie bisher die Objekte im Quellcode.

HINWEIS: Von der Möglichkeit, Objekte im XAML-Code zu instanziieren, halten die Autoren nicht allzu viel. Einerseits wird mit Klassen gearbeitet, die per Code definiert und verarbeitet werden, andererseits wird die Instanz in der Oberfläche, d.h. im XAML-Code, erzeugt. Das ist sicher nicht der Weisheit letzter Schluss. Gerade die üble Vermischung von Code und Oberfläche sollte eigentlich vermieden werden.

Fragwürdig werden Beispielprogramme dann, wenn im C#-Quellcode das zunächst in XAML erzeugte Objekt per *FindResource* gesucht wird (siehe folgender Abschnitt). Da pervertiert doch jede Form der sauberen Programmierung.

Wohlgemerkt wollen wir nicht die komplette Datenbindung im Code realisieren. Das ist sicher zu aufwändig und auch nicht notwendig. Doch aus Sicht des Programmierers sollte nicht die Oberfläche (XAML) sondern der Code im Mittelpunkt des Programms stehen.

15.2.1 Objekte im XAML-Code instanziieren

Erster Schritt, nach der Definition der Klasse, ist das Importieren des entsprechenden Namespaces in die XAML-Datei, andernfalls können Sie auch nicht darauf Bezug nehmen.

BEISPIEL 15.7: Import des aktuelle Namespace *Datenbindung* in die XAML-Datei

XAML

```
<Window x:Class="Datenbindung.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:Datenbindung"
...
```

Nachdem in XAML die entsprechende Klasse bekannt ist, kann diese auch verwendet werden, um eine eigene Instanz zu erzeugen.

BEISPIEL 15.8: Erzeugen der Instanz im XAML-Code (wir nutzen eine fiktive Klasse *Schüler*)

XAML

```
...
 <Window.Resources>
   <local:Schüler x:Key="sch1" Nachname="Gurkenkopf" Vorname="Siegfried" />
 </Window.Resources>
```

Die Werte im Einzelnen:

- *local*: Der Bezug auf den Namespace-Alias
- *Schüler:* Der Klassenname
- *x:Key*: Der Schlüssel unter dem die Instanz verwendet werden kann
- Nachname, Vorname: Das Setzen einzelner Eigenschaften für die Instanz von *Schüler*

Letzter Schritt: wir nutzen die Möglichkeiten der Datenbindung und binden zwei *TextBox*en an die Eigenschaften *Nachname* und *Vorname*.

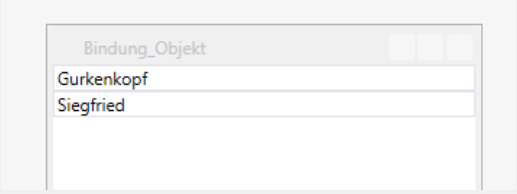
BEISPIEL 15.9: Bindung an das neue Objekt erzeugen

XAML

```
<StackPanel Name="StackPanel1">
  <TextBox Text="{Binding Source={StaticResource sch1}, Path=Nachname}" />
  <TextBox Text="{Binding Source={StaticResource sch1}, Path=Vorname}" />
</StackPanel>
```

Ergebnis

Schon zur Entwurfszeit dürfte in den beiden *TextBox*en der gewünschte Inhalt auftauchen:



Wem das zu viel Schreibarbeit ist, der kann mit dem *DataContext* auch eine alternative Variante der Zuweisung nutzen. Diese Eigenschaft bietet zunächst eine Alternativ zur Zuweisung von *Source*, hat jedoch zusätzlich die Fähigkeit, von übergeordneten auf untergeordnete Elemente vererbt zu werden. Damit können Sie beispielsweise einem *Panel* oder sogar dem gesamten *Window* einen *DataContext* zuweisen und diesen in allen enthaltenen Elementen nutzen.

BEISPIEL 15.10: Vereinfachung durch Verwendung eines *DataContext*

XAML

```
<StackPanel Name="StackPanel1" DataContext="{StaticResource sch1}">
  <TextBox Text="{Binding Path=Nachname}" />
  <TextBox Text="{Binding Path=Vorname}" />
...
```

Sie sparen sich so die Angabe von *Source* bei jedem einzelnen Element.

15.2.2 Verwenden der Instanz im C#-Quellcode

Sicher nicht ganz abwegig ist der Wunsch, zur Laufzeit per C#-Code auch mit dem Objekt zu arbeiten, um z.B. die Werte mit einer *MessageBox* anzuzeigen.

Hier wird die Programmierung dann schon recht windig, müssen Sie doch zunächst die entsprechende Ressource des *Window* suchen und typisieren, möglichen Fehlern ist Tür und Tor geöffnet.

BEISPIEL 15.11: Anzeige der Werte eines per XAML instanziierten Objekts

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Schüler mySch = (Schüler)FindResource("sch1");
    MessageBox.Show(mySch.Nachname + ", " + mySch.Vorname);
}
```

Aus Sicht eines Programmierers sieht das doch ziemlich merkwürdig aus, auch wenn sich hier der XAML-Profi freut, dass er sogar eine Instanz plus Wertzuweisung per XAML-Code realisiert hat.

Doch was passiert eigentlich mit der Datenbindung, wenn wir der Instanz ein paar neue Werte zuweisen? Ein Test ist schnell realisiert:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Schüler mySch = (Schüler)FindResource("sch1");
    mySch.Nachname = "Strohkopf";
}
```

Der nachfolgende Blick auf die Oberfläche dürfte in den meisten Fällen für Ernüchterung sorgen, haben Sie Ihre .NET-Klasse (in diesem Fall *Schüler*) nicht entsprechend angepasst, passiert überhaupt nichts und in den *TextBox*en stehen nach wie vor die alten Werte.

15.2.3 Anforderungen an die Quell-Klasse

Was ist hier schief gelaufen? Eigentlich nichts, die neuen Werte stehen wirklich im Objekt, werden aber nicht angezeigt, weil die darstellenden Elemente von einer Wertänderung nichts mitbekommen haben. Wir müssen diese quasi "wecken", und was eignet sich dafür besser als ein Ereignis?

Auch hier gibt es bereits eine fertige Lösung:

HINWEIS: Implementieren Sie in Ihrer Klasse das Interface *INotifyPropertyChanged* (Namespace *System.ComponentModel*).

BEISPIEL 15.12: Unsere Klasse *Schüler* mit implementiertem *NotifyPropertyChanged*-Ereignis

C#

```
using System.ComponentModel;

namespace Datenbindung
```

BEISPIEL 15.12: Unsere Klasse *Schüler* mit implementiertem *NotifyPropertyChanged*-Ereignis

```
{
    public class Schüler : INotifyPropertyChanged
    {
        string _Nachname;
        string _Vorname;
        DateTime _Geburtstag;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Vorname
        {
            get { return _Vorname; }
            set
            {
                _Vorname = value;
                NotifyPropertyChanged("Vorname");
            }
        }

        public string Nachname
        {
            get { return _Nachname; }
            set
            {
                _Nachname = value;
                NotifyPropertyChanged("Nachname");
            }
        }
...

        public override string ToString()
        {
            return this._Nachname + ", " + this._Vorname;
        }

        private void NotifyPropertyChanged(String info)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(info));
        }
    }
}
```

HINWEIS: Alternativ können Sie natürlich auch Abhängigkeitseigenschaften definieren, diese verfügen "ab Werk" über die erforderliche Benachrichtigung an die gebundenen Elemente, erfordern aber einen höheren Programmieraufwand.

HINWEIS: Damit die Klasse auch im XAML-Code instanziiert werden kann, muss diese über einen parameterlosen Konstruktor verfügen.

Einzige sinnvolle Ausnahme: Sie erzeugen per XAML Objekte und nutzen diese auch nur dort (z.B. Zugriff auf XML-Ressourcen per Url).

15.2.4 Instanziieren von Objekten per C#-Code

Eigentlich könnten wir Ihnen an dieser Stelle noch weitere Möglichkeiten zeigen, wie Sie in XAML Objekte erzeugen bzw. zuweisen können, aber dies ist weder sinnvoll noch besonders übersichtlich. Wir wollen uns stattdessen mit der Vorgehensweise bei vorhandenen, d h. per Code erzeugten, .NET-Objekten beschäftigen.

Zunächst bleiben wir bei unserem einfachen Beispiel mit der Instanz der Klasse *Schüler*.

BEISPIEL 15.13: Verwendung von instanziierten Objekten in XAML

C#

Zunächst die Instanziierung:

```
..
    public partial class Window1 : Window
    {
        public Schüler Schueler;

        public Window1()
        {
            InitializeComponent();
```

Instanz erzeugen und Werte zuweisen:

```
            Schueler = new Schüler { Nachname = "Mayer", Vorname = "Alexander",
                                     Geburtstag = new DateTime(2001, 11, 7) };
```

Hier legen wir per C#-Code den *DataContext* fest:

```
            StackPanel1.DataContext = Schueler;
        }
```

Die spätere Abfrage des Objekts stellt jetzt überhaupt kein Problem dar, die Instanz liegt ja bereits vor:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show(Schueler.Nachname + ", " + Schueler.Vorname);
        }
...
```

XAML

Der vollständige XAML-Code:

```
<Window x:Class="Datenbindung.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

BEISPIEL 15.13: Verwendung von instanziierten Objekten in XAML

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
 <StackPanel Name="StackPanel1">
   <TextBox Text="{Binding Path=Nachname}" />
   <TextBox Text="{Binding Path=Vorname}" />
   <Button Click="Button_Click">Prüfen</Button>
 </StackPanel>
</Window>
```

Der Vorteil dieser Vorgehensweise: Sie entscheiden, wie und wann die Instanz erzeugt wird, können vorher noch diverse Methoden aufrufen, profitieren von der Syntaxprüfung und haben einen lesbaren Code.

Der einzige Nachteil: Sie haben keine Wertanzeige zur Entwurfszeit, im XAML-Code ist es nicht sofort erkennbar, welches Objekt zugeordnet wird. Dies ist allerdings auch gleich wieder der Vorteil, mit einem Klick können Sie einen neuen *DataContext* zuweisen und eine andere Instanz bearbeiten.

15.3 Binden von Collections

Die bisherigen Ausführungen dürften zwar schon das Potenzial der Datenbindung demonstriert haben, doch nach der Pflicht kommt jetzt die Kür, d h. die Arbeit mit einer Reihe von Objekten (Collections). Diese sind vor allem dann interessant, wenn Sie Objekte von Datenbanken abrufen, um diese in Eingabedialogen oder gleich in Listenfeldern darzustellen. Ausgangspunkt können hier Geschäftsobjekte, LINQ-Abfragen etc. sein.

HINWEIS: Im vorliegenden Abschnitt werden wir uns zunächst auf eine "selbstgestrickte" Collection beziehen (wir verwenden das *Schüler*-Objekt aus dem vorhergehenden Abschnitt). Ab Seite 812 geht es dann mit Datenbindung in Verbindung mit LINQ to SQL-Abfragen weiter.

15.3.1 Anforderung an die Collection

Wie auch bei der Klassendefinition für das einzelne Objekt,werden auch an die Collection einige Anforderungen gestellt. Zwar können die WPF-Elemente durch die Verwendung der *INotifyPropertyChanged*-Schnittstelle auf Änderungen einzelner Objekteigenschaften reagieren, das Hinzufügen oder Löschen von ganzen Objekten ist davon aber nicht betroffen. Aus diesem Grund bietet WPF auch hier ein genormtes Interface für die Rückmeldung an: *INotifyCollectionChanged*.

HINWEIS: Grundvoraussetzung für die Anzeige von Listen ist die Verwendung des *IEnumerable*-Interfaces.

Wollen Sie es sich leicht machen, können Sie direkt Objekte der Klasse *ObservableCollection (*Namespace *System.Collections.ObjectModel*) erzeugen.

BEISPIEL 15.14: (Fortsetzung) Erzeugen und Verwenden einer geeigneten Klasse für die Datenbindung von Collections

C#

```
using System.Collections.ObjectModel;
...
    public partial class Objects_Collections : Window
    {
```

Eine Collection von Schülern:

```
        public ObservableCollection<Schüler> Klasse;
```

Im Konstruktor des *Window* erzeugen wir eine Instanz und füllen diese mit einigen Datensätzen:

```
        public Objects_Collections()
        {
            Klasse = new ObservableCollection<Schüler>();
            Klasse.Add(new Schüler { Nachname = "Mayer", Vorname = "Alexander",
                                     Geburtstag = new DateTime(2001, 11, 7) });
            Klasse.Add(new Schüler { Nachname = "Müller", Vorname = "Thomas",
                                     Geburtstag = new DateTime(2001, 10, 18) });
            Klasse.Add(new Schüler { Nachname = "Lehmann", Vorname = "Walter",
                                     Geburtstag = new DateTime(2001, 1, 21) });
            InitializeComponent();
```

Hier dürften Sie die Verbindung zur bisherigen Vorgehensweise sehen, die Collection wird als *DataContext* für das Fenster und damit für alle untergeordneten Elemente ausgewählt:

```
            this.DataContext = Klasse;
        }
```

15.3.2 Einfache Anzeige

Damit können wir uns zunächst der einfachen Anzeige, z.B. in *TextBox*en, widmen.

BEISPIEL 15.15: (Fortsetzung) Binden von TextBoxen an die Collection

XAML

```
  <StackPanel Grid.Column="1" Background="Aqua">
    <Label Content="Nachname:" />
```

Hier werden die *TextBox*en an die Eigenschaften der Collection bzw. an das aktive Objekt der Collection gebunden (dies ist durch eine View bestimmt, siehe ab Seite 808):

```
    <TextBox Name="txt1" Text="{Binding Path=Nachname}" />
    <Label Content="Vorname:" />
```

Beachten Sie auch diese mögliche Kurzsyntax, die auf die Angabe von *Path* verzichtet:

```
    <TextBox Name="txt2" Text="{Binding Vorname}" />
```

BEISPIEL 15.15: (Fortsetzung) Binden von TextBoxen an die Collection

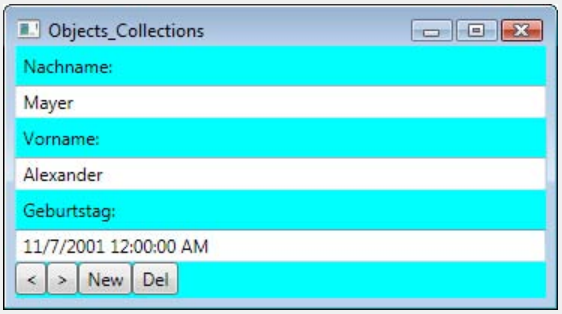
```
    <Label Content="Geburtstag:" />
    <TextBox Name="txt3" Text="{Binding Geburtstag}" />
```

Einige Schaltflächen definieren:

```
    <StackPanel Orientation="Horizontal">
      <Button Content=" &lt; " Click="Button_Click_1" />
      <Button Content=" &gt; " Click="Button_Click"/>
      <Button Content=" New " Click="Button_Click_2"/>
      <Button Content=" Del " Click="Button_Click_3"/>
    </StackPanel>
  </StackPanel>
```

Ergebnis

Das erzeugte Formular:



Nach dem Start dürfte schon etwas in den Textfeldern angezeigt werden, ein Navigieren zwischen den einzelnen Datensätzen (Objekten) ist allerdings noch nicht möglich.

15.3.3 Navigieren zwischen den Objekten

HINWEIS: An dieser Stelle müssen wir etwas vorgreifen, Abschnitt 15.4 ab Seite 808 geht auf dieses Thema im Detail ein.

Navigation zwischen Datensätzen bedeutet, dass auch irgendwo ein aktueller Datensatz gespeichert wird und entsprechende Navigationsmethoden zur Verfügung stehen. Auch bei intensiver Suche werden Sie aber derartige Eigenschaften zunächst nicht finden.

WPF erzeugt beim Binden von Collections automatisch eine Sicht auf die eigentliche Collection. Diese Sicht verwaltet den aktuellen Datensatz, bietet Navigationsmethoden an und ermöglicht das Filtern und Sortieren der Daten[1].

Diese automatisch erzeugte Sicht können Sie mit der Methode *CollectionViewSource.GetDefault-View* für eine spezifische Collection abrufen.

[1] Derartige Sichten können Sie auch selbst erstellen und quasi als Schicht zwischen Daten und DataContext schieben.

BEISPIEL 15.16: (Fortsetzung) Abrufen und Verwenden der *DefaultView* für unsere Collection

C#

Wir erweitern die Liste der lokalen Variablen, um die Sicht zu speichern:

```
        private ICollectionView view;
...
        public Objects_Collections()
        {
            Klasse = new ObservableCollection<Schüler>();
...
```

Im Konstruktor rufen wir die Sicht ab:

```
            this.view = CollectionViewSource.GetDefaultView(Klasse);
        }
```

Jetzt können wir mit dieser Sicht auch die Navigation zwischen den einzelnen Elementen der Collection realisieren.

Nächstes Objekt:

```
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            view.MoveCurrentToNext();
            if (view.IsCurrentAfterLast) view.MoveCurrentToLast();
        }
```

Vorhergehendes Objekt:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            view.MoveCurrentToPrevious();
            if (view.IsCurrentBeforeFirst) view.MoveCurrentToFirst();
        }
```

Wir fügen zum Testen ein neues Objekt zur Laufzeit in die Collection ein:

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            Klasse.Add(new Schüler { Nachname = "Möhre", Vorname = "Willi",
                                     Geburtstag = new DateTime(1919, 1, 1) });
        }
```

Auch das Löschen von Objekten ist auf diesem Wege möglich:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            Klasse.Remove(view.CurrentItem as Schüler);
        }
```

Nach dem Start des Beispiels können Sie zwischen den Objekten "navigieren", Objekte hinzufügen und diese auch wieder löschen. Das Ganze kommt Ihnen sicherlich unter dem Stichwort "Datenbanknavigator" bekannt vor.

15.3.4 Einfache Anzeige in einer ListBox

Das Anzeigen von Einzeldatensätzen ist ja schon ganz gut, wie aber steht es mit dem Füllen von ganzen Listenfeldern?

Auch hier können Sie, dank Datenbindung, schnell zu brauchbaren Ergebnissen kommen.

BEISPIEL 15.17: (Fortsetzung) Anbinden einer *ListBox* an unsere Collection

C#

Es genügt zunächst die einfache Zuweisung von "{Binding}" an die *ItemsSource:*

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox1"
         ItemsSource="{Binding}"/>
```

Der Hintergrund: Da die Collection bereits direkt an das Formular gebunden ist, brauchen wir hier nicht weitere Eigenschaften zu spezifizieren. Alternativ könnten Sie hier auch die Collection per *DataContext* zuweisen.

Und wofür ist das Attribut *IsSynchronizedWithCurrentItem* verantwortlich? Hier sollten Sie sich an unsere Sicht erinnern, die auch den aktuellen "Satzzeiger" verwaltet. Nur wenn Sie das Attribut auf *True* setzen, wird das aktuelle Item mit dem "Satzzeiger" synchronisiert (dies gilt für beide Richtungen).

Ergebnis

Die angezeigte *ListBox* zur Laufzeit:

HINWEIS: Die *ItemsSource*-Eigenschaft kann nur verwendet werden, wenn die *Items*-Collection eines *ItemsControl* leer ist. Falls nicht, wird Ihre Anwendung eine *InvalidOperationException* auslösen.

Doch woher "weiß" die *ListBox* eigentlich, welche Eigenschaften des *Schüler*-Objekts in der Liste darzustellen sind? Antwort: Sie weiß es nicht und verwendet in diesem Fall einfach die *ToString*-Methode des betreffenden Objekts. Wenn Sie jetzt mal kurz auf Seite 796 nachschlagen, werden Sie feststellen, dass wir in weiser Vorahnung bereits die *ToString*-Methode überschrieben haben und damit eine Kombination aus *Nachname* und *Vorname* zurückgeben (siehe oben).

Verwendung von DisplayMemberPath

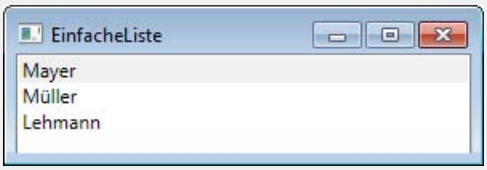
Natürlich ist das Überschreiben der *ToString*-Methode nicht der Weisheit letzter Schluss und so ist es sicher sinnvoll, noch einen anderen Weg zur Auswahl des anzuzeigenden Members zu unterstützen. Genau für diesen Zweck wird die *DisplayMemberPath*-Eigenschaft angeboten, diese bestimmt, welcher Member für den Text des Listeneintrags verwendet wird.

BEISPIEL 15.18: Verwendung von *DisplayMemberPath* für die Auswahl der anzuzeigenden Eigenschaft

XAML

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox1"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"/>
```

Ergebnis



Leider genügt jedoch auch diese Version der Anzeigeformatierung nicht immer und so landen wir unweigerlich bei den *DataTemplates*.

15.3.5 DataTemplates zur Anzeigeformatierung

Obige Art der Datenbindung dürfte in vielen Fällen wohl kaum genügen. Die WPF-Entwickler haben aber auch für für diesen Fall vorgesorgt und mit dem *DataTemplate* ein mächtiges Werkzeug geschaffen.

Das Prinzip: Jeder *ListBox/ComboBox* können Sie ein *DataTemplate* zuweisen, das dafür verantwortlich ist, wie das einzelne Item aufgebaut ist (quasi eine Schablone in die die Daten eingefügt werden). Und da WPF im Content eines Items fast jede Zusammenstellung von Elementen akzeptiert, können Sie hier Formatierungen beliebiger Art erzeugen (natürlich im Rahmen der XAML-Vorgaben).

BEISPIEL 15.19: (Fortsetzung) Wir wollen in der *ListBox* eine zweispaltige Anzeige realisieren (links der Nachname, rechts der Nachname und der Vorname).

XAML

In den Ressourcen (z.B. Window) erzeugen Sie das erforderliche *DataTemplate*:

```
<Window.Resources>
  <DataTemplate x:Key="SchülerListTemplate">
```

Das Layout bestimmen Sie:

```
    <StackPanel Orientation="Horizontal">
```

Bei der Zuweisung von Inhalten können Sie jetzt direkt auf die Eigenschaften zugreifen:

```
      <TextBlock VerticalAlignment="Top" Width="100" Text="{Binding Path=Nachname}" />
      <StackPanel>
        <TextBlock Text="{Binding Path=Nachname}" />
        <TextBlock Text="{Binding Path=Vorname}" />
      </StackPanel>
    </StackPanel>
  </DataTemplate>
</Window.Resources>
...
```

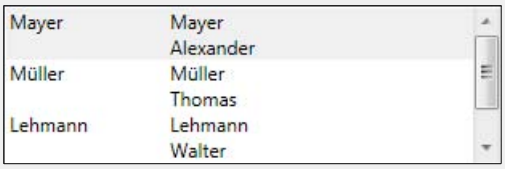
BEISPIEL 15.19: (Fortsetzung) Wir wollen in der *ListBox* eine zweispaltige Anzeige realisieren (links der Nachname, rechts der Nachname und der Vorname).

Last but not least, müssen Sie der *ListBox* auch noch das Template zuweisen:

```
<ListBox Height="100"  IsSynchronizedWithCurrentItem="True" Name="listBox2"
         ItemsSource="{Binding}" ItemTemplate="{StaticResource SchülerListTemplate}"/>
```

Ergebnis

Die erzeugte *ListBox:*



Dass Sie hier auch mit Grafiken, optischen Effekten, KontextMenüs etc. arbeiten können, sollte nach den Darstellungen der vorhergehenden Kapitel klar sein.

15.3.6 Mehr zu List- und ComboBox

An dieser Stelle wollen wir uns noch einige spezielle Eigenschaften von *List-* und *ComboBox* ansehen, die in der täglichen Programmierpraxis von Bedeutung sind.

SelectedIndex

Möchten Sie Einträge in der *ListBox* auswählen bzw. bestimmen, der wievielte Eintrag (Index) in der Liste markiert ist, können Sie die *SelectedIndex*-Eigenschaft verwenden.

BEISPIEL 15.20: Auswahl des zweiten Eintrags

C#

```
private void Button1_Click(object sender, RoutedEventArgs e)
{
    listBox1.SelectedIndex = 1;
}
```

SelectedItem/SelectedItems

Möchten Sie das markierte Listenelement selbst abrufen bzw. das damit verbundene Objekt, verwenden Sie die *SelectedItem*-Eigenschaft. Alternativ können Sie auch eine Liste der markierten Einträge mit *SelectedItems* abrufen.

HINWEIS: Die Collection *SelectedItems* steht Ihnen nur zur Verfügung, wenn Sie *SelectionMode* auf *Multiple* festgelegt haben.

BEISPIEL 15.21: Verwendung *SelectedItem*/*SelectedItems*

C#

Wir nutzen unsere überschriebene *ToString*-Methode:

```
MessageBox.Show(listBox1.SelectedItem.ToString());
```

Wir greifen direkt auf einen Member (typisieren nicht vergessen) zu:

```
MessageBox.Show((listBox1.SelectedItem as Schüler).Nachname);
```

Wir zeigen alle markierten Einträge an:

```
foreach (Schüler s in listBox1.SelectedItems)
    MessageBox.Show(s.Nachname);
```

SelectedValuePath und SelectedValue

Mit *SelectedValuePath* können Sie festlegen, welcher Member von der Eigenschaft *SelectedValue* zurückgegeben wird. Dies ist im Zusammenhang mit Datenbanken meist der Primärindex der Tabelle, mit dem Sie einen Datensatz eindeutig identifizieren können.

HINWEIS: Ist *SelectedValuePath* nicht festgelegt, gibt *SelectedValue* das komplette Objekt zurück (entspricht *SelectedItem*).

BEISPIEL 15.22: Verwendung *SelectedValuePath* und *SelectedValue*

XAML

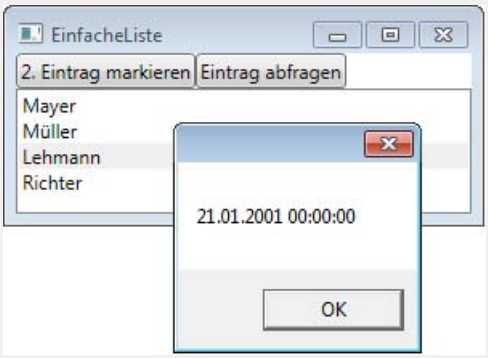
```
<ListBox  IsSynchronizedWithCurrentItem="True" Name="listBox1"
          ItemsSource="{Binding}" DisplayMemberPath="Nachname"
          SelectedValuePath="Geburtstag"/>
```

C#

```
private void Button2_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show(listBox1.SelectedValue.ToString());
}
```

Ergebnis

Das Ergebnis zur Laufzeit:



15.3.7 Verwendung der ListView

Im vorhergehenden Kapitel hatten wir die *ListView* ja bereits kurz gestreift (Trockenschwimmen), an dieser Stelle zeigen wir Ihnen die *ListView* "in Action".

Einfache Bindung

Prinzipiell ist die *ListView* der *ListBox* recht ähnlich, die Anbindung der Einträge erfolgt ebenfalls per *ItemsSource,* die Auswahl bzw. Bestimmung (*SelectedItem, SelectedValue* etc.) der markierten Einträge ist analog realisiert.

Neu ist, das die *ListView* über Spaltenköpfe verfügt, die Sie getrennt konfigurieren können (*GridViewColumnHeader*) und gegebenenfalls auch für das Sortieren (siehe 2. Beispiel) verwenden können.

Ein weiterer Unterschied ist die Unterstützung von verschiedenen Ansichten, von denen jedoch nur die *GridView* vordefiniert ist. Im weiteren werden wir uns auch nur auf diese Ansicht beschränken.

BEISPIEL 15.23: (Fortsetzung) Anzeige der Collection-Daten in einer *ListView*

XAML

Zuweisen der Datenquelle (Übernahme von *Window.DataContext*):

```
<ListView Height="100" IsSynchronizedWithCurrentItem="True" ItemsSource="{Binding}">
  <ListView.View>
```

Hier wird die *GridView* definiert:

```
    <GridView>
```

Die einzelnen Spalten definieren:

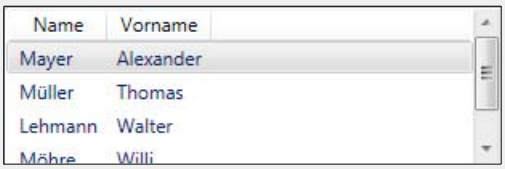
```
      <GridView.Columns>
```

Und jetzt wird es einfach, binden Sie lediglich die gewünschten Eigenschaften an die einzelnen Spalten der *GridView:*

```
        <GridViewColumn Header="Name" DisplayMemberBinding="{Binding Path=Nachname}" />
        <GridViewColumn Header="Vorname"
                        DisplayMemberBinding="{Binding Path=Vorname}" />
      </GridView.Columns>
    </GridView>
  </ListView.View>
</ListView>
```

Ergebnis

Das Endergebnis zur Laufzeit:



Sortieren der Einträge

Wie schon erwähnt, können Sie die Spaltenköpfe auch für das Sortieren der Einträge nutzen. Ein einfaches Beispiel zeigt die Vorgehensweise:

BEISPIEL 15.24: Sortieren nach Klick auf den jeweiligen Spaltenkopf

XAML

Unsere Änderung in der Seitenbeschreibung:

```
<ListView Name="ListView1" IsSynchronizedWithCurrentItem="True"
          ItemsSource="{Binding}">
    <ListView.View>
        <GridView>
            <GridView.Columns>
                <GridViewColumn DisplayMemberBinding="{Binding Path=Nachname}" >
                    <GridViewColumnHeader Click="SortClick" Content="Nachname" />
                </GridViewColumn>
                <GridViewColumn DisplayMemberBinding="{Binding Path=Vorname}" >
                    <GridViewColumnHeader Click="SortClick" Content="Vorname" />
                </GridViewColumn>
            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>
```

C#

Der Quellcode fällt recht kurz aus:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
...
       private void SortClick(object sender, RoutedEventArgs e)
        {
```

Zunächst die betreffende Spalte bestimmen:

```
            GridViewColumnHeader spalte = sender as GridViewColumnHeader;
```

Die Defaultview bestimmen:

```
            ICollectionView view =
                          CollectionViewSource.GetDefaultView(ListView1.ItemsSource);
```

Eine neue Sortierfolge festlegen:

```
            view.SortDescriptions.Clear();
            view.SortDescriptions.Add(new SortDescription(spalte.Content.ToString(),
                                      ListSortDirection.Ascending));
```

Und aktualisieren:

```
            view.Refresh();
        }
```

Auf weitere Experimente mit der *ListView* verzichten wir an dieser Stelle, mit dem *DataGrid* steht uns ein wesentlich mächtigeres Control zur Verfügung. Mehr dazu ab Seite 825.

Wie Sie auch größere Collections bändigen, zeigt das Praxisbeispiel

▶ 15.9.1 Collections in Hintergrundthreads füllen

15.4 Noch einmal zurück zu den Details

Nachdem wir in den bisherigen Abschnitten schon mehrfach vorgreifen mussten, wollen wir an dieser Stelle noch einmal kurz auf einige Details der Datenbindung eingehen.

Interessant für den Datenbankprogrammierer ist vor allem eine Zwischenschicht, die vom WPF quasi zwischen die Daten (Collections) und die reinen Anzeige-Controls (z.B. *ListView*) geschoben wird, um einige datenbanktypische Operationen zu ermöglichen:

- Verwaltung des aktuellen Satzzeigers
- Navigation zwischen den Datensätzen
- Sortierfunktion
- Filterfunktion

Die Rede ist von der Klasse *CollectionView*, um deren Erzeugung Sie sich nicht selbst kümmern müssen, da Sie diese automatisch erstellte View recht einfach abrufen können.

BEISPIEL 15.25: Abrufen der *CollectionView*

C#

```
...
        NWDataContext db = new NWDataContext();
        ICollectionView view0;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            lvOrder.DataContext = db.Orders;
            view0 = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

Mit dieser *CollectionView* stellt es jetzt kein Problem mehr dar, die oben gewünschten Datenbankfunktionen zu implementieren.

15.4.1 Navigieren in den Daten

Wie schon in den vorhergehenden Abschnitten gezeigt, ist eine der Hauptaufgaben der *CollectionView* die Verwaltung des "Satzzeigers". Dazu steht Ihnen zunächst die Eigenschaft *CurrentItem* zur Verfügung, die das aktuell ausgewählte Element der gebundenen Collection zurückgibt.

Weitere interessante Eigenschaften:

Eigenschaften	Beschreibung
CurrentItem	Aktuelles Element der Auflistung.
CurrentPosition	Ordinalposition des aktuellen Elements in der Auflistung.
IsCurrentAfterLast	Befindet sich der "Satzzeiger" hinter dem Ende der Auflistung?
IsCurrentBeforeFirst	Befindet sich der "Satzzeiger" vor dem Beginn der Auflistung?

Die eigentliche Navigation realisieren Sie mit den folgenden Methoden:

Methoden	Beschreibung
MoveCurrentTo	Das übergebene Element wird als *CurrentItem* festgelegt.
MoveCurrentToFirst	"Satzzeiger" auf das erste Element verschieben.
MoveCurrentToLast	"Satzzeiger" auf das letzte Element verschieben.
MoveCurrentToNext	"Satzzeiger" auf das folgende Element verschieben.
MoveCurrentToPosition	"Satzzeiger" auf den angegebenen Index verschieben.
MoveCurrentToPrevious	"Satzzeiger" auf das vorhergehende Element verschieben.

BEISPIEL 15.26: Navigationstasten für "Vor" und "Zurück"

C#

Der folgende Aufwand ist nötig, um nicht hinter bzw. vor dem letzten bzw. ersten Datensatz zu landen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToNext();
    if (view.IsCurrentAfterLast) view.MoveCurrentToLast();
}

private void Button_Click_1(object sender, RoutedEventArgs e)
{
    view.MoveCurrentToPrevious();
    if (view.IsCurrentBeforeFirst) view.MoveCurrentToFirst();
}
```

BEISPIEL 15.27: Verwendung von *CurrentItem*

C#

Löschen eines Listeneintrags per *CurrentItem* und Typisierung:

```
private void Button_Click_3(object sender, RoutedEventArgs e)
{
    Klasse.Remove(view.CurrentItem as Schüler);
}
```

15.4.2 Sortieren

Dass sich die *CollectionView* auch zum Sortieren eignet, haben wir ja bereits am Beispiel der *ListView* gezeigt, wo durch Klicken auf den Spaltenkopf die Collection nach der jeweiligen Spalte sortiert wurde.

Zum Einsatz kommt die Collection *SortDescriptions,* die neben den Membernamen auch die Sortierfolge enthält. Da es sich um eine Collection handelt, können Sie auch mehrere Elemente angeben:

BEISPIEL 15.28: Sortieren einer Collection

C#

```
private void SortClick(object sender, RoutedEventArgs e)
{
   GridViewColumnHeader spalte = sender as GridViewColumnHeader;
```

CollectionView abrufen:

```
    ICollectionView view = CollectionViewSource.GetDefaultView(ListView1.ItemsSource);
```

Bisherige Sortiervorgaben löschen:

```
    view.SortDescriptions.Clear();
```

Eine neue Sortierfolge (Spaltenname, Aufsteigend) festlegen:

```
    view.SortDescriptions.Add(new SortDescription(spalte.Content.ToString(),
                              ListSortDirection.Ascending));
```

Ansicht aktualisieren:

```
    view.Refresh();
}
```

15.4.3 Filtern

Auch wenn Sie mit dieser Variante vorsichtig sein sollten (Daten werden eigentlich vor der Anzeige gefiltert, um unnötigen Traffic zu vermeiden), so besteht doch die Möglichkeit, zur Laufzeit gezielt Daten aus der gebundenen Collection herauszufiltern. Nutzen Sie dazu die *Filter*-Eigenschaft, der Sie eine selbst zu definierende Methode zuweisen.

BEISPIEL 15.29: Filter festlegen

C#

Zunächst unsere Filterfunktion (alle Einträge die mit "T" beginnen):

```
protected bool MeinFilter(object value)
{
    Schüler s = value as Schüler;
    return s.Vorname.StartsWith("T");
}
```

BEISPIEL 15.29: Filter festlegen

Und hier wird der Filter zugewiesen (ein Aktualisieren ist nicht nötig):

```
private void Button3_Click(object sender, RoutedEventArgs e)
{
    ICollectionView view = CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
    view.Filter += MeinFilter;
}
```

HINWEIS: Möchten Sie den Filter wieder löschen, weisen Sie der Eigenschaft einfach *null* zu.

15.4.4 Live Shaping

Die vorhergehenden Funktionen zum Sortieren, Filtern und auch Gruppieren funktionieren recht gut, auch wenn Sie zum Beispiel neue Einträge zur Collection hinzufügen. Alternativ können Sie auch die *Refresh*-Methode der *CollectionView* aufrufen.

Doch was, wenn Sie lediglich ein Objekt der Collection bearbeiten und sich so z.B. die Filterbedingung für dieses Objekt ändert? In diesem Fall werden Sie schnell feststellen, das Anzeige und Inhalt der Collection nicht mehr übereinstimmt.

BEISPIEL 15.30: (Fortsetzung) Fehlende Aktualisierung

C#

```
...
```

Filtern Sie die Daten und rufen Sie folgende Methode auf, passiert nichts:

```
private void Button5_Click(object sender, RoutedEventArgs e)
{
    Schüler schueler = Klasse.Where(sch => sch.Vorname == "Thomas").FirstOrDefault();
    schueler.Vorname = "aaaa";
```

Erst nach einem Refresh sind die gefilterten Daten auch aktuell:

```
    //ICollectionView view = CollectionViewSource.GetDefaultView(
                                                        listBox1.ItemsSource);
    //view.Refresh();
}
...
```

Hier hilft Ihnen Live Shaping weiter. Über das Interface *ICollectionViewLiveShaping* können Sie bestimmte Spalten zur Überwachung anmelden.

Jeweils drei neue Member sind für die weitere Arbeit interessant:

- Die Eigenschaft *CanChangeLiveFiltering* (*... Sorting, ...Grouping*) bestimmt, ob die Überwachung möglich ist.
- Die Eigenschaft *IsLiveFiltering* (*... Sorting*, *...Grouping*) bestimmt, ob die Überwachung eingeschaltet ist

- Die Collection *LiveFilteringProperties* (*... Sorting...*, *...Grouping...*) enthält die Namen der zu überwachenden Eigenschaften.

BEISPIEL 15.31: Filtern mit Live Shaping

C#

```
...
        private void Button6_Click(object sender, RoutedEventArgs e)
        {
```

Filter wie bekannt festlegen:

```
            ICollectionView view = CollectionViewSource.GetDefaultView(listBox1.ItemsSource);
            view.Filter += MeinFilter;
```

Wir rufen das neue Interface ab:

```
            ICollectionViewLiveShaping viewls = view as ICollectionViewLiveShaping;
```

Test auf Interface:

```
            if (viewls == null)
            {
                MessageBox.Show("Nicht unterstützt!");
            }
```

Ist die Überwachung möglich:

```
            if (viewls.CanChangeLiveFiltering)
            {
```

Feld *Vorname* soll überwacht werden:

```
                viewls.LiveFilteringProperties.Add("Vorname");
                viewls.IsLiveFiltering = true;
            }
        }
...
```

HINWEIS: Da diese Form der Überwachung recht ressourcenintensiv ist, sollten Sie davon nur Gebrauch machen, wenn es unbedingt nötig ist.

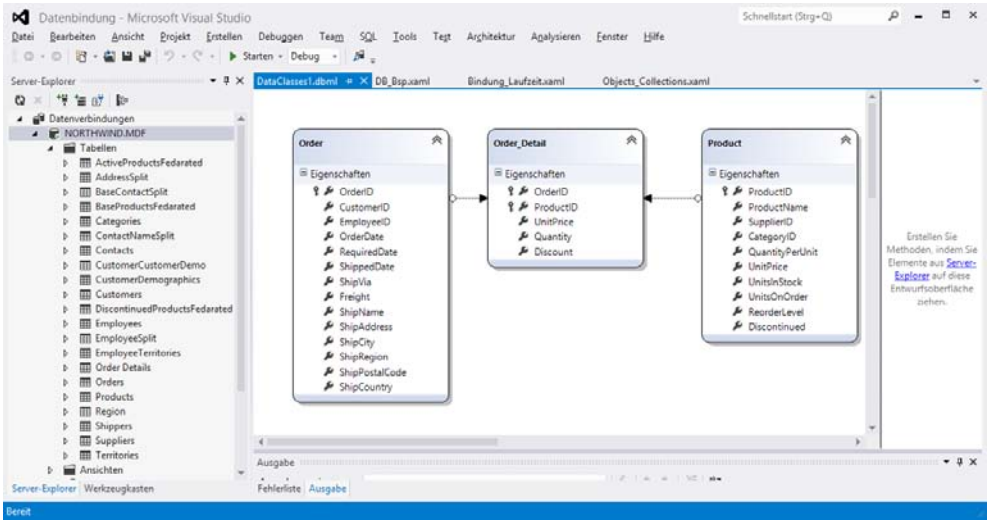
15.5 Anzeige von Datenbankinhalten

Anzeige eigener Collections gut und schön, aber wir wollen auch noch kurz einen Blick aufs große Ganze werfen und damit sind wir schon bei der "Königsdisziplin", den Datenbanken, angelangt.

HINWEIS: Wer jetzt Berge von ADO.NET-Quellcode erwartet, den werden wir enttäuschen. Für den Zugriff auf unsere *Northwind*-Beispieldatenbank werden wir zunächst LINQ to SQL verwenden, den kleinen Bruder des Entity Frameworks. Mehr dazu finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012].

15.5.1 Datenmodell per LINQ to SQL-Designer erzeugen

Fügen Sie Ihrem WPF-Projekt eine neue "LINQ to SQL Klasse" hinzu, um den LINQ to SQL-Designer zu öffnen (*Projekt|Neues Element hinzufügen*). Damit haben Sie bereits die zentrale *DataContext*-Klasse[1] erstellt. Den Namen dieser Klasse können Sie jetzt gegebenenfalls über das Eigenschaftenfenster anpassen (wir wählen *NWDataContext*).



In die noch leere Arbeitsfläche (diese ähnelt dem Klassendesigner) fügen Sie die benötigten SQL-Server-Tabellen ein. Nutzen Sie dazu den Server-Explorer (siehe linke Seite).

HINWEIS: Für unser Beispiel fügen Sie die Tabellen *Order, Order_Detail* und *Product* ein.

Der Designer erstellt nachfolgend automatisch die erforderlichen C#-Mapperklassen für die einzelnen Tabellen sowie deren Associations. Damit sind wir aber auch schon wieder bei den schon bekannten Collections angekommen, die weitere Vorgehensweise dürfte Ihnen also bekannt vorkommen.

HINWEIS: Sie können neben reinen Tabellen auch Views bzw. Gespeicherte Prozeduren in den Designer einfügen. Views werden wie Tabellen behandelt, Gespeicherte Prozeduren werden als Methoden der *DataContext*-Klasse mit typisierten Rückgabewerten gemappt.

[1] Ja das ist wieder eine der "glücklich" gewählten Namensübereinstimmungen. Dieser *DataContext* hat nichts mit dem WPF-DataContext zu tun!

15.5.2 Die Programm-Oberfläche

Nach Schließen des Designers wollen wir uns mit einem einfachen WPF-Projekt von der Funktionsfähigkeit überzeugen.

```
<Window x:Class="Datenbindung.DB_Bsp"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Eine Ereignisprozedur beim Öffnen des Window:

```
    Title="DB_Bsp" Height="300" Width="476" Loaded="Window_Loaded">
```

Zweispaltiges Layout per *Grid*:

```
 <Grid>
   <Grid.ColumnDefinitions>
     <ColumnDefinition Width="75" />
     <ColumnDefinition Width="*" />
   </Grid.ColumnDefinitions>
```

Hier die *ListView* mit den vorhandenen Bestellungen (Tabelle *Order*)*:*

```
   <ListView Grid.Column="0" Name="lvOrder" IsSynchronizedWithCurrentItem="True"
             ItemsSource="{Binding}" SelectionChanged="lvOrder_SelectionChanged"
             HorizontalAlignment="Left" >
```

Die eigentlich Bindung erfolgt per *DataContext*-Zuweisung im C#-Code. Über das *SelectionChanged*-Ereignis werden wir die Detaildaten in die zweite *ListView* "zaubern".

```
     <ListView.View>
       <GridView>
         <GridView.Columns>
```

Wir zeigen nur die Spalte mit der Bestellnummer an:

```
           <GridViewColumn Header="OrderID" DisplayMemberBinding="{Binding OrderID}" />
         </GridView.Columns>
       </GridView>
     </ListView.View>
   </ListView>
```

Die *ListView* für die Detaildaten (die *DataContext*-Eigenschaft setzen wir im *SelectionChanged*-Ereignis der obigen *ListView*):

```
   <ListView Grid.Column="1" Name="lvOrderDetails" IsSynchronizedWithCurrentItem="True"
             ItemsSource="{Binding}" >
     <ListView.View>
       <GridView>
         <GridView.Columns>
           <GridViewColumn Header="OrderId" DisplayMemberBinding="{Binding OrderID}" />
           <GridViewColumn Header="ID" DisplayMemberBinding="{Binding ProductID}" />
```

Wer jetzt erwartet, dass die Autoren sich die Mühe machen und noch eine dritte *GridView* für die Artikelnamen einbinden, hat nicht mit der Leistungsfähigkeit von LINQ to SQL gerechnet. Es genügt die Abfrage der untergeordneten Collection *Product*:

```
          <GridViewColumn Header="Artikelname"
                          DisplayMemberBinding="{Binding Product.ProductName}" />
```

Ein sinnvoller Vorteil von objekt-relationalen Mapper-Klassen!

```
        </GridView.Columns>
      </GridView>
    </ListView.View>
  </ListView>
 </Grid>
</Window>
```

15.5.3 Der Zugriff auf die Daten

Jetzt müssen wir noch den erforderlichen C#-Code erstellen, um die Daten auch aus der Datenbank abzurufen.

```
...
    public partial class DB_Bsp : Window
    {
```

Die meiste Arbeit nimmt uns der LINQ to SQL-DataContext ab, den wir gleich zu Beginn instanziieren:

```
        NWDataContext db = new NWDataContext();
```

Wir wollen auch die Default-View zwischenspeichern, da wir diese für das Abrufen der Detaildatensätze benötigen:

```
        ICollectionView viewO;
...
```

Beim Laden des Fensters:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Den *DataContext* der linken *ListView* wir die Tabelle *Orders* zugewiesen:

```
            lvOrder.DataContext = db.Orders;
```

Die Default-View abrufen:

```
            viewO = CollectionViewSource.GetDefaultView(lvOrder.DataContext);
        }
```

So, das wäre schon alles, wenn da nicht noch die Detaildatenanzeige fehlen würde. Im *SelectionChanged*-Ereignis der linken *ListView* kümmern wir uns zunächst um das Abrufen des aktuellen Datensatzes und leiten aus diesem Objekt die OrderDetails ab (als Collection enthalten):

```
        private void lvOrder_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            lvOrderDetails.DataContext = (view0.CurrentItem as Order).Order_Details;
        }
    }
```

Wer jetzt beim Zugriff auf den MS SQL Server noch Wert auf ADO.NET-Objekte legt, dem ist nicht zu helfen. Kürzer kann das Beispiel kaum ausfallen.

Ach ja, wie kommen eigentlich neue Datensätze in die Tabellen? Hier genügt es, wenn Sie z.B. ein neues *Product*-Objekt erstellen und es an die *Products*-Collection anhängen. Mit einen *Submit-Changes* des *DataContext*-Objekts (der LINQ to SQL *DataContext*) ist die Änderung dann auch schon zum Server übertragen.

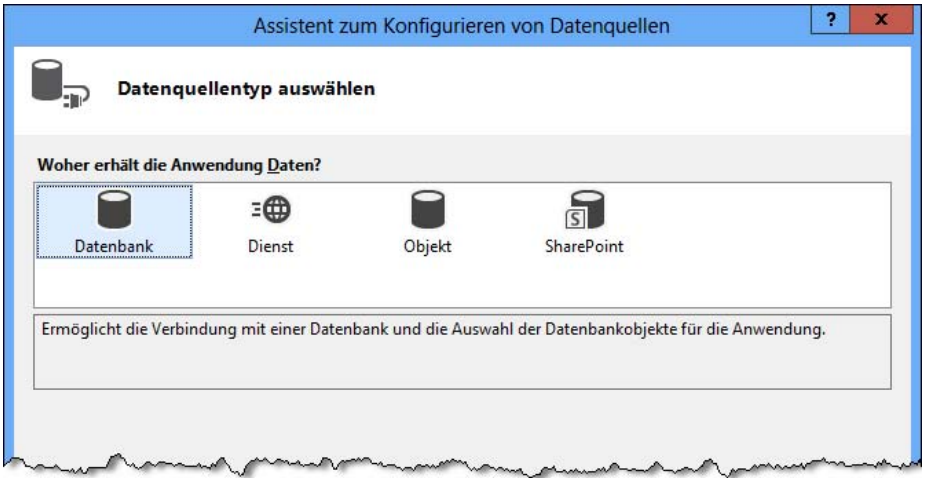
15.6 Drag & Drop-Datenbindung

Die in den vorhergehenden Abschnitten gezeigte Vorgehensweise war recht einfach und mit wenig Schreibaufwand verbunden. Wenn Sie hier weiter lesen, sind Sie vermutlich daran interessiert, noch weniger Code zu produzieren und stattdessen die Assistenten für sich arbeiten zu lassen. Mal sehen, ob Sie mit dem Ergebnis und der Vorgehensweise zufrieden sind!

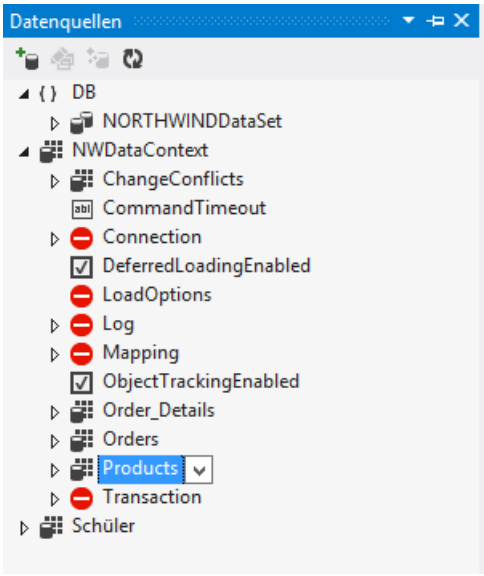
15.6.1 Vorgehensweise

Die Vorgehensweise orientiert sich an der Arbeitsweise bei den Windows Forms:

- Öffnen Sie das Datenquellen-Fenster.
- Ist die gewünschte Datenquelle noch nicht vorhanden, erzeugen Sie diese über den Klick auf den "Hinzufügen"-Button. In diesem Fall sollte jetzt der Datenquellen-Assistent erscheinen:

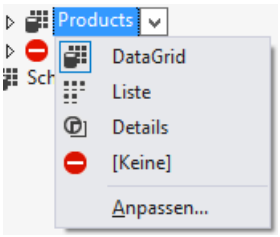


- Wählen Sie den Eintrag "Datenbank", wenn Sie eine Datenbank gänzlich neu einfügen wollen, oder "Objekt", wenn Sie bereits über ein Datenmodell (z.B. LINQ to SQL) verfügen.
- Haben Sie die Datenquelle bzw. die Datenobjekte erfolgreich eingebunden, dürften diese im Datenquellen-Fenster angezeigt werden:

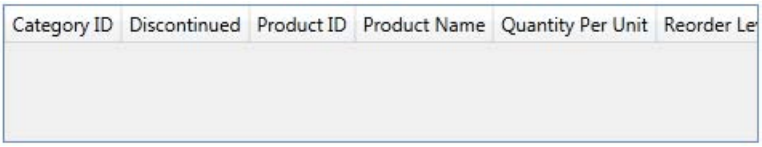


Obige Abbildung zeigt ein eingebundenes DataSet, einen LINQ to SQL-DataContext und unsere *Schüler*-Klasse.

- Wählen Sie jetzt beispielsweise eine Collection "Products" im Datenquellen-Fenster aus, wird Ihnen folgende Auswahlliste angezeigt:

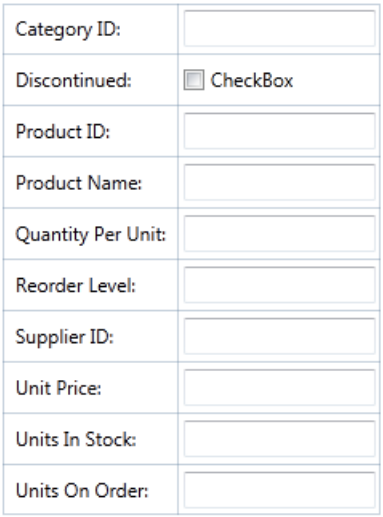


Die Auswahl bestimmt, welche Controls nach einer Drag&Drop-Operation mit dieser Collection in das Fenster eingefügt werden. Wählen Sie die erste Option wird automatisch ein komplett fertig konfiguriertes *DataGrid* in Ihr WPF-Formular eingefügt:



Analog gilt dieses auch für die Auswahl "List", hier kommt eine *ListView* zum Einsatz, die jedoch, auf Templates aufbauend, statt statischem Text Textfelder verwendet.

Mit der Auswahl "Details" wird Ihnen ein *Grid* mit den erforderlichen *Label-, TextBox-* und *CheckBox*-Controls generiert:



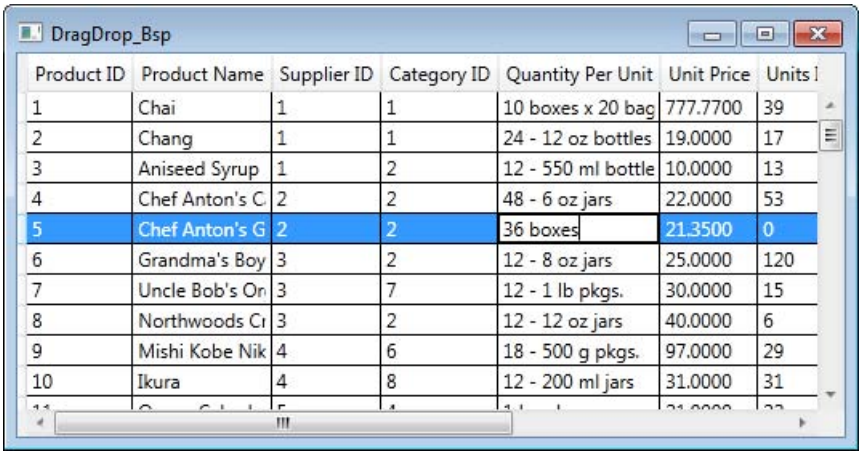
- Abschließend sollten Sie ein Blick auf den C#-Quellcode des Formulars werfen, hier ist teilweise noch mit etwas Arbeit zu rechnen:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            Datenbindung.DB.NORTHWINDDataSet NORTHWINDDataSet =
```

```
        ((Datenbindung.DB.NORTHWINDDataSet)(this.FindResource("NORTHWINDDataSet")));
        // Lädt Daten in Tabelle "Products". Sie können diesen Code nach Bedarf ändern.
        Datenbindung.DB.NORTHWINDDataSetTableAdapters.ProductsTableAdapter
        nORTHWINDDataSetProductsTableAdapter = new
          Datenbindung.DB.NORTHWINDDataSetTableAdapters.ProductsTableAdapter();
        nORTHWINDDataSetProductsTableAdapter.Fill(NORTHWINDDataSet.Products);
        System.Windows.Data.CollectionViewSource productsViewSource =
              ((System.Windows.Data.CollectionViewSource)
             (this.FindResource("productsViewSource")));
        productsViewSource.View.MoveCurrentToFirst();
    }
```

Im obigen Beispiel (Anbindung eines DataSets) brauchen Sie keine Änderung vorzunehmen, bei der Anbindung von LINQ to SQL-DataContext-Objekten müssen Sie sich jedoch selbst um das Erstellen des *DataContext* kümmern.

- Einem Probelauf des Programm steht jetzt nichts mehr im Wege:



15.6.2 Weitere Möglichkeiten

Selbstverständlich können Sie jetzt noch den XAML-Code nachbearbeiten und Spalten ein-/ausblenden bzw. umformatieren. Dazu genügt es meist, wenn Sie ein neues *DataTemplate* mit dem gewünschten Eingabe-Control zuweisen:

Statt einer einfachen *TextBox*:

```
<DataGridTextColumn Binding="{Binding Path=OrderDate}" Header="Order-Date"
                    Width="SizeToHeader" />
```

können Sie zum Beispiel auch einen Kalender einblenden:

```
<DataGridTemplateColumn Header="Order Date" Width="SizeToHeader">
    <DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
```

```
                        <DatePicker SelectedDate="{Binding Path=OrderDate}" />
                    </DataTemplate>
                </DataGridTemplateColumn.CellTemplate>
            </DataGridTemplateColumn>
```

Dass Sie auch alle anderen Formatierungsmöglichkeiten des *DataGrid* nutzen können, brauchen wir an dieser Stelle sicher nicht zu erwähnen.

Interessant für den Programmierer ist noch der automatisch erstellte *<Window.Resources>*-Abschnitt, in dem sowohl das nötige *DataSet* als auch eine *CollectionViewSource* erzeugt wird:

```
<Window x:Class="Datenbindung.DragDrop_Bsp"
...
    xmlns:my1="clr-namespace:Datenbindung">
    <Window.Resources>
        <my:NORTHWINDDataSet x:Key="NORTHWINDDataSet" />
        <CollectionViewSource x:Key="productsViewSource"
              Source="{Binding Path=Products, Source={StaticResource NORTHWINDDataSet}}" />
    </Window.Resources>
```

Letztere können Sie dazu nutzen, um zum Beispiel eine Navigation zwischen den Datensätzen zu realisieren:

Instanz ermitteln:

```
...
     System.Windows.Data.CollectionViewSource productsViewSource =
       ((System.Windows.Data.CollectionViewSource)(this.FindResource("productsViewSource")));
```

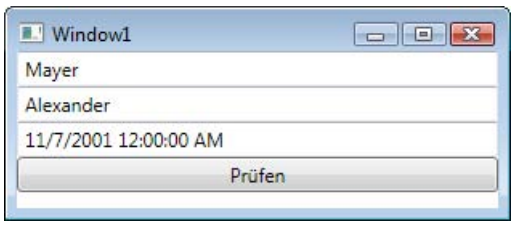
Über die *View* stehen Ihnen alle Navigationsmöglichkeiten zur Verfügung:

```
     productsViewSource.View.MoveCurrentToFirst();
```

15.7 Formatieren von Werten

In unseren Beispielen haben wir uns bislang erfolgreich davor gedrückt, Datumswerte, Währungen etc. in einem sinnvollen Format anzuzeigen bzw. zu formatieren.

Binden Sie beispielsweise einen Datumswert an eine *TextBox,* wird zunächst das Standardformat angezeigt:



Das sieht aus deutscher Sicht zunächst wenig erfreulich aus, aber mit dem *Language*-Attribut können Sie hier etwas nachhelfen.

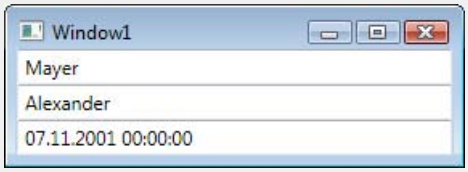
BEISPIEL 15.32: Verwendung *Language*-Attribut

XAML

```
<TextBox Text="{Binding Path=Geburtstag}" Language="de"/>
```

Ergebnis

Nachfolgend sollte zumindest ein deutscher Datumswert angezeigt werden:



Doch auch dies ist noch nicht der Weisheit letzter Schluss.

15.7.1 IValueConverter

Mit Hilfe der WPF-Wertkonvertierer können Sie jede beliebige Konvertierung zwischen Quelle und Ziel einer Datenbindung realisieren. Dazu erstellen Sie eine Klasse, die das *IValueConverter*-Interface unterstützt. Diese Klasse muss zwei Methoden implementieren:

- *Convert* (von der Quelle zum Ziel)
- *ConvertBack* (vom Ziel zur Quelle)

Sicher können Sie sich denken, dass die *ConvertBack*-Methode den höheren Programmieraufwand erfordert, hat doch hier der User die Möglichkeit, zunächst beliebige Werte in die Textfelder einzugeben, die Sie dann mühsam in den geforderten Datentyp umwandeln müssen.

BEISPIEL 15.33: Implementieren und Verwenden eines Wert-Konvertierers

C#

An dieser Stelle wollen wir allerdings nicht das Rad neu erfinden, sondern ein Beispiel aus dem Microsoft MSDN darstellen.

Zunächst der eigentliche Wert-Konvertierer:

```
using System.Globalization;

namespace Datenbindung
{
    public partial class Window1 : Window
    {
```

Hier die neue Klasse *DateConverter,* die Sie mit entsprechenden Attributen versehen sollten:

```
        [ValueConversion(typeof(DateTime), typeof(String))]
        public class DateConverter : IValueConverter
```

BEISPIEL 15.33: Implementieren und Verwenden eines Wert-Konvertierers

```
        {
```

Konvertieren von der Quelle zum Ziel (übergeben werden die Quelleigenschaft, der Zieleigenschaft-Typ, ein Konverter-Parameter sowie die aktuellen Landeseinstellungen):

```
            public object Convert(object value, Type targetType, object parameter,
                                  CultureInfo culture)
            {
                DateTime date = (DateTime)value;
                return date.ToShortDateString();
            }
```

Konvertieren vom Ziel (z.B. *TextBox*) zur Quelle (z.B. Objekt):

```
            public object ConvertBack(object value, Type targetType, object parameter,
                                      CultureInfo culture)
            {
                string strValue = value.ToString();
                DateTime resultDateTime;
                if (DateTime.TryParse(strValue, out resultDateTime))
                {
                    return resultDateTime;
                }
                return value;
            }
        }
```

XAML

Die Verwendung im XAML-Code:

```
<Window x:Class="Datenbindung.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Zunächst den lokalen Namespace einbinden:

```
    xmlns:local="clr-namespace:Datenbindung"
    Title="Window1" Height="300" Width="300" >
```

Die Einbindung der Klasse erfolgt per Ressource:

```
 <Window.Resources>
   <local:DateConverter x:Key="dateConverter"/>
 </Window.Resources>
 <StackPanel Name="StackPanel1">
   <TextBox Text="{Binding Path=Nachname}" Name="txt1" />
   <TextBox Text="{Binding Path=Vorname}" />
```

Und hier verwenden wir den Konverter bei der Bindung:

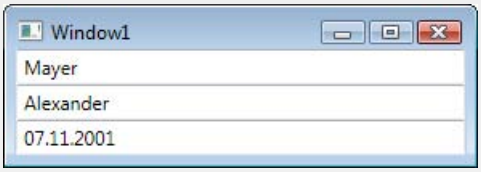
```
   <TextBox Text="{Binding Path=Geburtstag, Converter={StaticResource dateConverter}}" />
   <Button Click="Button_Click">Prüfen</Button>
```

BEISPIEL 15.33: Implementieren und Verwenden eines Wert-Konvertierers

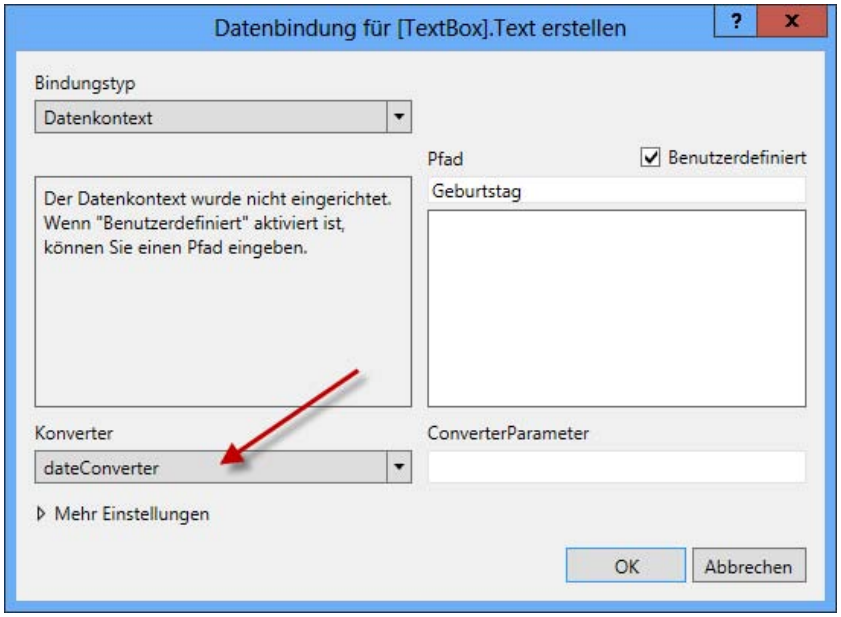
```
 </StackPanel>
</Window>
```

Ergebnis

Das neue Ergebnis sieht schon viel ansprechender aus:



HINWEIS: Zusätzlich besteht die Möglichkeit, vorhandene Wertkonvertierer per Eigenschaften-editor (siehe folgende Abbildung) zuzuweisen. Der Eigenschafteneditor erstellt, falls erforderlich, die entsprechenden Einträge im *<Window.Resources>*-Abschnitt des Formulars und weist das Attribut "Converter" zu.



15.7.2 BindingBase.StringFormat-Eigenschaft

Nachdem Sie sich durch unser obiges Beispiel gequält haben, wollen wir Ihnen auch nicht die dritte Variante zur Formatierung von Werten vorenthalten.

Werfen Sie doch einmal einen Blick auf die *BindingBase.StringFormat*-Eigenschaft, welche die Verwendung eines *IValueConverters* in vielen Standardfällen überflüssig macht.

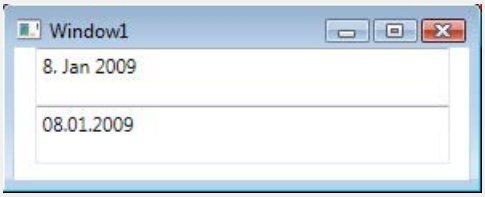
BEISPIEL 15.34: Zwei verschiedene Datumsformate zuweisen

XAML

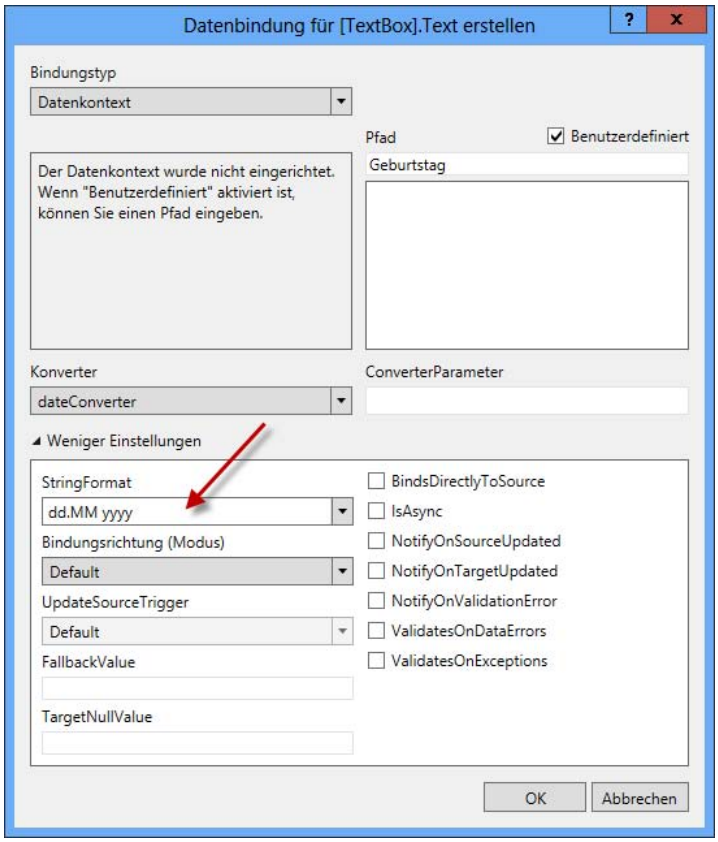
```
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= d. MMM yyyy}" />
...
<TextBox Text="{Binding Path=Geburtstag, StringFormat= dd.MM.yyyy }" />
...
```

Ergebnis

Das Ergebnis:



HINWEIS: Alternativ können Sie auch hier den Eigenschafteneditor nutzen (siehe folgende Abbildung), der eine einfache Zuweisung des Formatierungsstrings erlaubt.



15.8 Das DataGrid als Universalwerkzeug

Seit der WPF-Version 4 wird auch ein *DataGrid* regulär unterstützt, ohne zusätzliche Toolkits etc. laden zu müssen. Wie die schon besprochene *ListView* erlaubt auch das *DataGrid* die Anzeige von Collections im Tabellenformat. Zusätzlich werden Funktionen zum Editieren, Löschen, Auswählen und Sortieren angeboten.

HINWEIS: Anhand einiger Fallbeispiele wollen wir Ihnen ein Übersicht des Funktionsumfangs geben, für eine komplette Beschreibung aller Eigenschaften bzw. Möglichkeiten fehlt hier jedoch der Platz und wir verweisen auf die recht umfangreiche Hilfe zum *DataGrid*-Control.

15.8.1 Grundlagen der Anzeige

Wie fast nicht anders zu erwarten, erfolgt die Anbindung an die Datenquelle mittels *ItemsSource*-Eigenschaft, wir erzählen Ihnen an dieser Stelle also nichts Neues und verweisen auf die vorhergehenden Abschnitte.

Im Unterschied zu den bereits beschriebenen Controls bietet uns das *DataGrid* einen wesentlichen Vorteil, Sie brauchen sich nicht um das Erstellen der einzelnen Spalten zu kümmern, dank *AutoGenerateColumns*-Eigenschaft werden automatisch die erforderlichen Spalten erzeugt.

BEISPIEL 15.35: Anbinden des *DataGrid* an LINQ to SQL-Daten

XAML

```
<DataGrid Name="DataGrid1" />
```

C#

```
public partial class DataGrid_Bsp : Window
{
    NWDataContext db = new NWDataContext();

    public DataGrid_Bsp()
    {
        InitializeComponent();
        DataGrid1.ItemsSource = db.Products;
    }
```

Ergebnis

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit
1	Chai	1	1	10 boxes x 20 bags
2	Chang	1	1	24 - 12 oz bottles
3	Aniseed Syrup	1	2	12 - 550 ml bottles
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars
5	Chef Anton's Gumbo Mix	2	2	36 boxes
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars

Die Verwendung der *AutoGenerateColumns*-Eigenschaft ist sicher recht praktisch, doch haben Sie in diesem Fall keinen Einfluss auf Anzahl, Reihenfolge und Aussehen der Spalten.

Das *DataGrid* selbst unterscheidet in diesem Fall lediglich zwischen Spalten der Typen *DataGrid-TextColumn* und *DataGridCheckBoxColumn*, deren Bedeutung sich bereits durch den Namen erklärt.

15.8.2 UI-Virtualisierung

Sicher interessiert es Sie auch, wie leistungsfähig das *DataGrid* ist. Erstellen Sie ruhig einmal eine Collection mit 1.000.000 Datensätzen und weisen Sie diese als *ItemsSource* zu. Sie werden feststellen, dass das Erzeugen der Collection wesentlich länger dauert als die Anzeige der Daten. Der Grund für dieses Verhalten basiert auf der UI-Virtualisierung, die mit Hilfe eines *VirtualizingStackPanel* als Layoutpanel innerhalb des *DataGrid* (auch *ListView*, *ListBox* etc.) verwendet wird.

Das *VirtualizingStackPanel* sorgt dafür, dass nur die gerade sichtbaren Einträge (bzw. die dazu notwendigen Controls) erzeugt werden. Was passiert, wenn dies nicht so ist, können Sie ganz einfach ausprobieren. Es genügt, wenn Sie das folgende Attribut in die Elementedefinition einfügen:

```
<DataGrid VirtualizingStackPanel.IsVirtualizing="False" Name="DataGrid1" />
```

Bitte besorgen Sie sich rechtzeitig eine Zeitung und eine Kanne Kaffee wenn Sie versuchen wollen, eine große Collection an das *DataGrid* zu binden. Im extremsten Fall kommt es zur Meldung, dass der verfügbare Arbeitsspeicher nicht ausreicht. Die Ursache dürfte schnell klar werden, wenn Sie sich vorstellen, dass für jede erforderliche Zeile und alle angezeigten Spalten die entsprechenden Anzeige-Controls generiert werden müssen.

15.8.3 Spalten selbst definieren

Gehen Ihnen die Möglichkeiten von *AutoGenerateColumns* nicht weit genug, können Sie alternativ auch selbst Hand anlegen und die einzelnen Spalten frei definieren. Setzen Sie in diesem Fall das Attribut *AutoGenerateColumns* auf *false* und fügen Sie die Spaltendefinitionen der *Columns*-Eigenschaft hinzu (die Reihenfolge der Definition entscheidet über die Anzeigereihenfolge).

Wir machen es uns in diesem Fall zunächst etwas einfacher und generieren das *DataGrid* mit allen Spaltendefinitionen per Drag&Drop-Datenbindung (siehe Seite 816).

BEISPIEL 15.36: *DataGrid* mit einzeln definierten Spalten

XAML

Zunächst das Erzeugen der *CollectionViewSource*:

```
...
        <CollectionViewSource x:Key="schülerViewSource" d:DesignSource=
                              "{d:DesignInstance my:Schüler, CreateList=True}" />
...
```

Hier das *DataGrid*:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
```

BEISPIEL 15.36: *DataGrid* mit einzeln definierten Spalten

```
            ItemsSource="{Binding}" Name="DataGrid1"
            RowDetailsVisibilityMode="VisibleWhenSelected" >
```

Und hier folgen die Definitionen der einzelnen Spalten:

```
        <DataGrid.Columns>
```

Eine Textspalte erzeugen, Bindung an den Member *Nachname* herstellen, die Kopfzeile mit "Nachname" beschriften und eine Größenanpassung vornehmen:

```
            <DataGridTextColumn x:Name="nachnameColumn"
              Binding="{Binding Path=Nachname}" Header="Nachname" Width="SizeToHeader" />
```

Gleiches für den Vornamen:

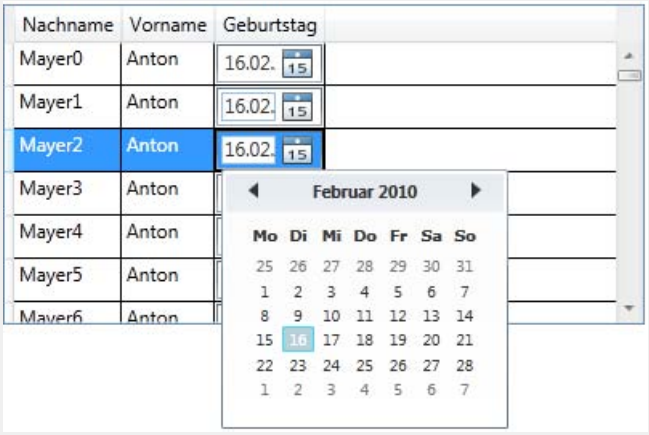
```
            <DataGridTextColumn x:Name="vornameColumn" Binding="{Binding Path=Vorname}"
                                Header="Vorname" Width="SizeToHeader" />
```

An dieser Stelle war der Assistent schon ganz schön "pfiffig", statt einer einfachen Textspalte hat er bereits ein *DataTemplate* mit einem *DatePicker* für die Datumsanzeige erzeugt:

```
            <DataGridTemplateColumn x:Name="geburtstagColumn"
                                    Header="Geburtstag" Width="SizeToHeader">
                <DataGridTemplateColumn.CellTemplate>
                    <DataTemplate>
                        <DatePicker SelectedDate="{Binding Path=Geburtstag}" />
                    </DataTemplate>
                </DataGridTemplateColumn.CellTemplate>
            </DataGridTemplateColumn>
        </DataGrid.Columns>
    </DataGrid>
</DockPanel>
```

Ergebnis

Das erzeugte *DataGrid*:



BEISPIEL 15.36: *DataGrid* mit einzeln definierten Spalten

Anmerkung

Im obigen Beispiel ist die Datumsspalte noch zu schmal, weisen Sie einfach der Eigenschaft *Width* einen größeren Wert zu:

```
<DataGridTemplateColumn x:Name="geburtstagColumn" Header="Geburtstag" Width="100">
```

HINWEIS: Sie können die Spaltenbreite auch mit "*" angeben, in diesem Fall verwendet die Spalte den restlichen verfügbaren Platz.

Wie Sie gesehen haben, steht Ihnen neben den Standard-Spaltentypen

- *DataGridTextColumn,*
- *DataGridCheckBoxColumn,*
- *DataGridComboBoxColumn,*
- *DataGridHyperlinkColumn*

auch die recht flexible *DataGridTemplateColumn* zur Verfügung. Welche Controls Sie hier einbinden (*Image, Chart*, *RichtextBox* etc.) bleibt Ihrer Phantasie überlassen.

Weitere Gestaltungsmöglichkeiten bieten sich mit dem Ein- und Ausblenden der Trennlinien, der Konfiguration der Spaltenköpfe per Template usw.

15.8.4 Zusatzinformationen in den Zeilen anzeigen

Nicht alle Informationen sollen immer gleich in einem Grid sichtbar sein, vielfach werden Detailfenster etc. eingeblendet, um nach der Auswahl eines Datensatzes weitere Informationen anzuzeigen. An dieser Stelle bietet das *DataGrid* mit dem *RowDetailsTemplate* ein recht interessantes Feature, versetzt Sie dieses Template doch in die Lage, unter bestimmten Umständen (*RowDetailsVisibilityMode*-Eigenschaft) zusätzliche Inhalte einzublenden.

BEISPIEL 15.37: Verwendung von *RowDetailsTemplate*

XAML

Zunächst müssen Sie bestimmen, wann die Details eingeblendet werden sollen:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
            ItemsSource="{Binding}" RowDetailsVisibilityMode="VisibleWhenSelected" >
            <DataGrid.Columns>
...
            </DataGrid.Columns>
```

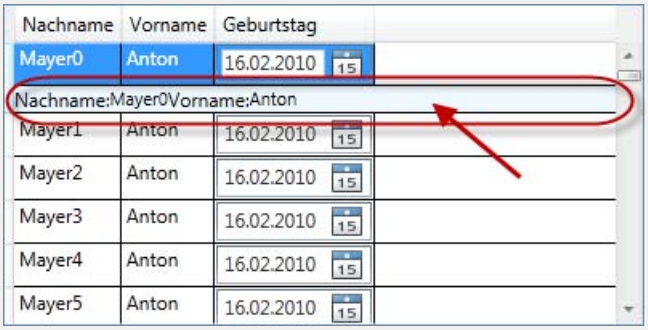
Nach der Spaltendefinition können Sie dann das *RowDetailsTemplate* einfügen und mit den gewünschten Informationen füllen:

```
            <DataGrid.RowDetailsTemplate>
```

BEISPIEL 15.37: Verwendung von *RowDetailsTemplate*

```
            <DataTemplate>
                <StackPanel Orientation="Horizontal" Background="AliceBlue">
                    <TextBlock>Nachname: </TextBlock>
                    <TextBlock Text="{Binding Path=Nachname}" FontSize="11" />
                    <TextBlock>  Vorname: </TextBlock>
                    <TextBlock Text="{Binding Path=Vorname}" FontSize="11" />
                </StackPanel>
            </DataTemplate>
        </DataGrid.RowDetailsTemplate>
    </DataGrid>
```

Ergebnis



Mit *RowDetailsVisibilityMode* bestimmen Sie, wie die Zeilendetails angezeigt werden. Standardwert ist *Collapsed* (nicht sichtbar) alternativ steht *Visible* (immer sichtbar) oder *VisibleWhenSelected* zur Verfügung (nur die aktuelle Zeile).

15.8.5 Vom Betrachten zum Editieren

Auch wenn die umfangreichen Anzeigeoptionen das *DataGrid* für diverse Aufgaben prädestinieren, eine Hauptaufgabe dürfte in den meisten Fällen auch das Editieren der Inhalte sein.

Grundsätzlich entscheidet zunächst die übergreifende Eigenschaft *IsReadOnly* über die Fähigkeit, Inhalte des *DataGrid*s zu editieren oder nur zu betrachten. Gleiches gilt auch auf Spaltenebene, auch hier können Sie mit *IsReadOnly* darüber entscheiden, welche Spalten editierbar sind und welche nicht. Zusätzlich unterstützen Sie diverse Ereignisse vor, während und nach dem Editiervorgang (*BeginningEdit*, *PreparingCellForEdit, CellEditEnding* ...).

HINWEIS: An dieser Stelle wollen wir uns jedoch aus Platzgründen vom *DataGrid* verabschieden und verweisen auf unser [Visual C# 2012 Kochbuch].

15.9 Praxisbeispiele

15.9.1 Collections in Hintergrundthreads füllen

In den vorhergehenden Beispielen haben wir es uns recht einfach gemacht. Eine Collection wurde erzeugt, gefüllt und angezeigt. Soweit so gut, aber was, wenn das Erzeugen der Collection etwas länger dauert? Ein kleines Beispielprogramm zeigt das Problem und natürlich auch die Lösung dafür. Dabei trennen wir aber zwischen der bisherigen Lösung und einer mit .NET 4.5 eingeführten Neuerung.

Oberfläche

Ein einfaches *Window* mit einigen Schaltflächen und einer *ListView* zur Anzeige der Daten:

```
<Window x:Class="Datenbindung.Laden_im_Hintergrund"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Laden_im_Hintergrund" Height="300" Width="530">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Button Click="Button_Click_2">Laden Vordergrund-Thread</Button>
            <Button Click="Button_Click_3">Laden Hintergrund-Thread</Button>
            <Button Click="Button_Click_4">Laden Hintergrund Lösung</Button>
            <Button Click="Button_Click_1">Laden neu</Button>
        </StackPanel>
        <ListView Name="ListView1" IsSynchronizedWithCurrentItem="True"
                  ItemsSource="{Binding}" VirtualizingPanel.IsVirtualizing="True" />
    </DockPanel>
</Window>
```

Das Problem

Stellen Sie sich folgendes Szenario vor: Sie füllen eine Liste von *Schüler*-Objekten[1], leider dauert der Abruf jedes einzelnen Objekts etwas länger:

```
    public partial class Laden_im_Hintergrund : Window
    {
        public ObservableCollection<Schüler> _klasse;

        public Laden_im_Hintergrund()
        {
            InitializeComponent();
            _klasse = new ObservableCollection<Schüler>();
            this.DataContext = _klasse;
        }
```

[1] Definition siehe Seite 795.

```
        private void Datenabrufen()
        {
            _klasse.Clear();
            for (int i = 0; i < 100; i++)
            {
```

Hier simulieren wir eine Zeitverzögerung, z.B. eine langsame Datenverbindung:

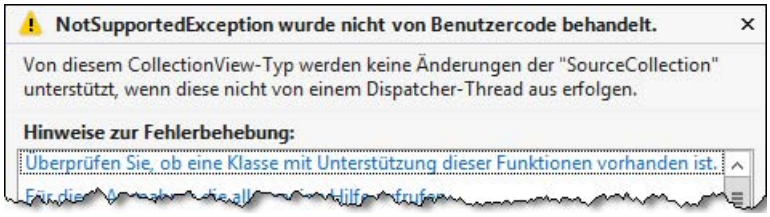
```
                Thread.Sleep(100);
                _klasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
                            Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
            }
        }

        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            Datenabrufen();
        }
```

Starten Sie die Anwendung, werden Sie nach einem Klick auf die Schaltfläche feststellen, dass Ihre Anwendung "einfriert". Diese Lösung wollen Sie dem Endanwender sicher nicht zumuten. Was liegt also näher, als diese Aufgabe in einen Hintergrundthread zu verlagern.

Gesagt, getan, wir kapseln obigen Methodenaufruf in einem extra Thread:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            Task.Factory.StartNew(Datenabrufen);
        }
```

Doch nach einem Start der Anwendung werden Sie schnell wieder auf den Boden zurückgeholt:



Sie können auf die Collection nicht per Hintergrundthread zugreifen. Das ist erstmal ein Show-Stopper. Doch es gibt zwei Lösungen:

- Laden einer extra Collection im Hintergrund und kopieren dieser Collection in den Vordergrund. Nachfolgend Abgleich mit der gebundenen Collection.
- Laden der Daten im Hintergrund, Einfügen der einzelnen Einträge durch jeweiligen Wechsel in den Vordergrundthread.

Die zweite Lösung ist mit häufigen Threadwechseln verbunden, wir sehen uns also die erste Lösung näher an.

Lösung (bis .NET 4.0)

Wir lagern das Laden der Daten in eine Funktion aus, die eine komplette Liste zurückgibt:

```
private ObservableCollection<Schüler> Datenabrufen_alteLoesung()
{
    ObservableCollection<Schüler> _threadklasse = new ObservableCollection<Schüler>();
    for (int i = 0; i < 100; i++)
    {
        Thread.Sleep(100);  // simuliert Laden aus der Quelle
        _threadklasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
         Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
    }
    return _threadklasse;
}
```

Unser Aufruf:

```
private void Button_Click_4(object sender, RoutedEventArgs e)
{
```

Anzeige, dass der Nutzer warten soll:

```
    this.Cursor = Cursors.Wait;
```

In einem extra Thread werden die Daten geladen:

```
    Task.Factory.StartNew <ObservableCollection<Schüler>>(
                            Datenabrufen_alteLoesung).ContinueWith(t =>
    {
```

Ist dies erfolgt, kopieren wir diese in die gebundene Liste:

```
        _klasse.Clear();
        foreach (var s in t.Result)
            _klasse.Add(s);
```

Und blenden die Sanduhr aus:

```
        this.Cursor = null;
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Test

Nach dem Start wird die "Sanduhr" angezeigt, nach einigen Sekunden ist die Liste gefüllt. Wie Sie sehen, muss sich der Nutzer auch hier gedulden, die Oberfläche bleibt in dieser Zeit aber voll bedienbar.

Lösung (ab .NET 4.5)

Neu seit .NET 4.5 ist die Möglichkeit, Collections für die gleichzeitige Bearbeitung in Threads quasi anzumelden. Nutzen Sie dazu einen Aufruf der Methode *EnableCollectionSynchronization*.

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Anmelden der Collection, wir übergeben noch das aktuelle Window-Objekt als Sperrobjekt[1]:

```
            BindingOperations.EnableCollectionSynchronization(_klasse, this);
```

Wir rufen die Daten per extra Thread ab:

```
            Task.Factory.StartNew(Datenabrufen);
        }

        private void Datenabrufen()
        {
            _klasse.Clear();
            for (int i = 0; i < 100; i++)
            {
                Thread.Sleep(100);  // simuliert Laden aus der Quelle
                _klasse.Add(new Schüler { Nachname = "Mayer" + i.ToString(),
                 Vorname = "Alexander", Geburtstag = new DateTime(2001, 11, 7) });
            }
        }
```

Test

Nach dem Start werden Sie feststellen, dass die Oberfläche beweglich bleibt, und dass die Daten "tröpfchenweise" in die Liste geladen werden , Sie können beim Füllen quasi zusehen. Eine einfache und recht elegante Lösung für das Einlesen größerer Datenmengen.

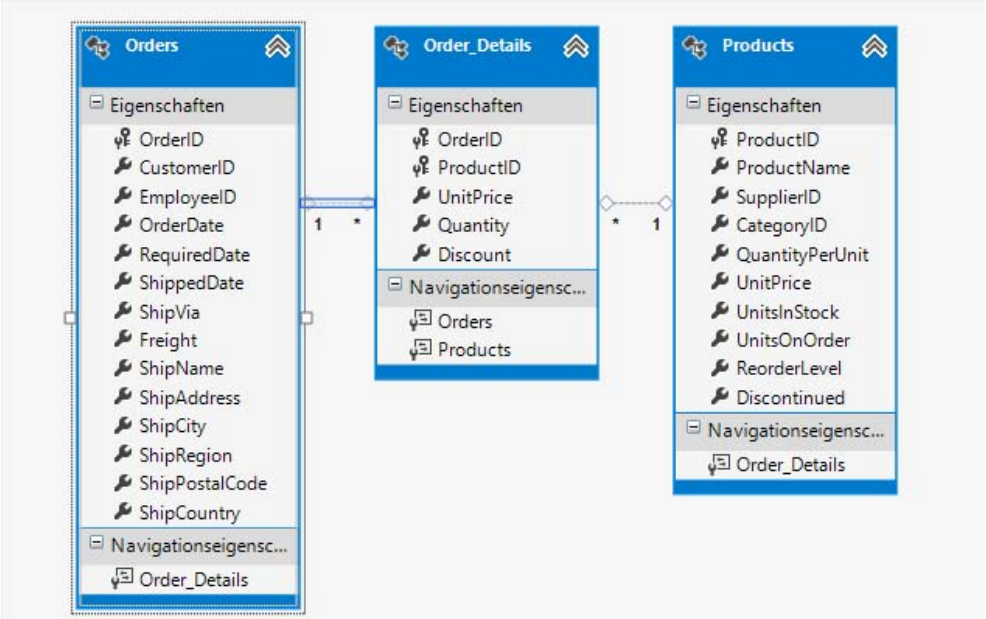
15.9.2 Drag & Drop-Bindung bei 1:n-Beziehungen

Grundlage unseres kleinen Beispiels soll in diesem Fall ein Entity Data Model sein, bei dem wir die Beziehung zwischen den Tabellen *Orders* und *Order_Details* mit zwei *DataGrids* visualisieren wollen. Wir machen es uns jetzt jedoch einfach und verwenden für den Entwurf das Datenquellen-Fenster, so kommen wir ohne eine einzige Zeile eigenen Quellcodes aus.

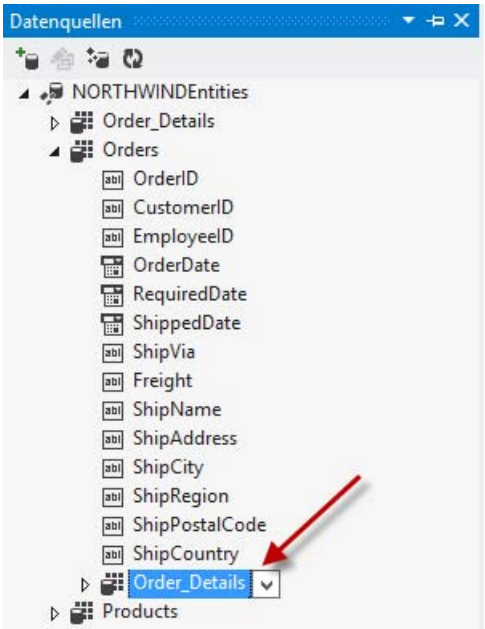
Oberfläche

Erstellen Sie zunächst ein neues WPF-Projekt, in das Sie die Datenbank *Northwind.mdf* einfügen. Bei der Frage nach dem zu erzeugenden Datenbankmodell wählen Sie *Entity Data Model*, um ein entsprechendes Modell zu erzeugen. Bei der Auswahl der Tabellen können Sie sich auf die Tabellen *Orders*, *Order_Details* und *Products* beschränken.

[1] Es tut auch jede andere Objekt-Instanz.



Nachfolgend wenden Sie sich dem Entwurf der Oberfläche zu. Ziehen Sie aus dem Datenquellen-Fenster den Eintrag *Orders* direkt in die Freifläche des WPF-Formulars. Automatisch wird jetzt ein *DataGrid* für die Anzeige der Bestellungen erzeugt (siehe Laufzeitansicht). Für die Anzeige der Bestellungsdetails ziehen Sie einfach den untergeordneten Knoten *Order_Details* ebenfalls in das WPF-Formular.



HINWEIS: Verwechseln Sie diesen Knoten nicht mit dem Knoten *Order_Details*, der unabhängig im DataContext definiert ist!

Damit ist unser Programm bereits "komplett", wir wollen jedoch noch kurz einen Blick auf den generierten Quellcode werfen, um die Funktionsweise besser zu verstehen.

Quellcode (XAML)

Bei der Oberflächendefinition interessieren wir uns nur für die beiden *DataGrid*s:

```
<Window x:Class="EDM_WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Master/Detail-Beziehung" Height="456" Width="630" mc:Ignorable="d"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Einbinden des aktuellen Namespace und Zuweisen einer Ereignismethode in der die Daten geladen werden:

```
       xmlns:my="clr-namespace:EDM_WPF" Loaded="Window_Loaded">
    <Window.Resources>
```

Für jedes der beiden DataGrids wird eine *CollectionViewSource* erzeugt:

```
        <CollectionViewSource x:Key="ordersViewSource"
                              d:DesignSource="{d:DesignInstance my:Orders, CreateList=True}" />
        <CollectionViewSource x:Key="ordersOrder_DetailsViewSource"
                              Source="{Binding Path=Order_Details,
                              Source={StaticResource ordersViewSource}}" />
    </Window.Resources>
```

Den Datenkontext zentral zuweisen, beide *DataGrid*-Objekte befinden sich im *DockPanel* und erben deshalb diese Eigenschaft (Achtung: hier werden die Details zugewiesen):

```
    <DockPanel DataContext="{StaticResource ordersOrder_DetailsViewSource}">
```

Das Master-*DataGrid* an die entsprechende *CollectionViewSource* binden:

```
        <DataGrid DockPanel.Dock="Top" AutoGenerateColumns="False"
EnableRowVirtualization="True"
            Height="188" ItemsSource="{Binding Source={StaticResource ordersViewSource}}"
            Name="ordersDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected">
...
        </DataGrid>
```

Das Detail-*DataGrid* nutzt die im *DockPanel* definierte Bindung:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding}"
        Name="order_DetailsDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected" >
...
```

```
        </DataGrid>
    </DockPanel>
</Window>
```

HINWEIS: Obiger Code ist sicher nicht der Weisheit letzter Schluss, besser Sie räumen hier etwas auf und löschen die *DataContext*-Zuweisung beim *DockPanel* und ändern die Definition des Detail-DataGrids wie folgt:

```
        <DataGrid AutoGenerateColumns="False" EnableRowVirtualization="True"
                  ItemsSource="{Binding Source={StaticResource ordersOrder_DetailsViewSource}}"
                  Name="order_DetailsDataGrid" RowDetailsVisibilityMode="VisibleWhenSelected" >
```

Quellcode (C#)

Auch der automatisch erzeugte Quellcode scheint noch verbesserungswürdig:

```
    public partial class MainWindow : Window
    {
...
```

Mit dem Laden des Formulars werden die Daten geladen:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            EDM_WPF.NORTHWINDEntities nORTHWINDEntities = new EDM_WPF.NORTHWINDEntities();
```

Die *CollectionViewSource*-Instanz (Master) aus der Oberfläche abrufen:

```
            System.Windows.Data.CollectionViewSource ordersViewSource =
              ((System.Windows.Data.CollectionViewSource)
(this.FindResource("ordersViewSource")));
```

Hier wird zunächst die Datenabfrage definiert (siehe Hilfsmethode):

```
            System.Data.Objects.ObjectQuery<EDM_WPF.Orders> ordersQuery =
               this.GetOrdersQuery(nORTHWINDEntities);
```

Abrufen der Daten und zuweisen an die *CollectionViewSource*:

```
            ordersViewSource.Source =
                  ordersQuery.Execute(System.Data.Objects.MergeOption.AppendOnly);
        }
```

Folgende Hilfsmethode definiert die Abfrage für die Master- **und** die Detail-Tabelle:

```
        private System.Data.Objects.ObjectQuery<Orders> GetOrdersQuery(
                                                        NORTHWINDEntities nORTHWINDEntities)
        {
            System.Data.Objects.ObjectQuery<EDM_WPF.Orders> ordersQuery =
                     nORTHWINDEntities.Orders;
```

Hier schließen wir die Detaildaten in die Abfrage ein:

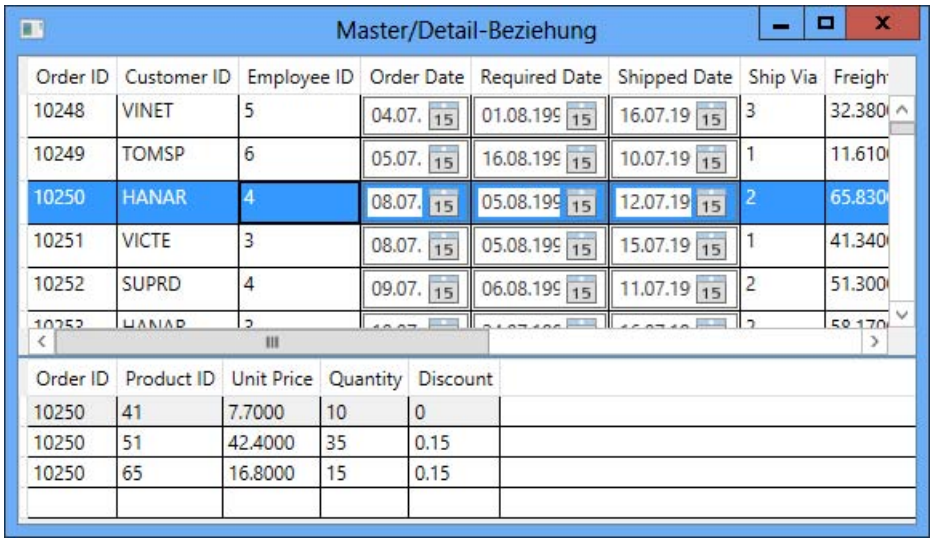
```
            ordersQuery = ordersQuery.Include("Order_Details");
            return ordersQuery;
        }
    }
```

Sie können den Quellcode durchaus noch etwas vereinfachen: Löschen Sie die Methode *GetOrdersQuery* und ändern Sie das *Window_Loaded*-Ereignis wie folgt:

```
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            EDM_WPF.NORTHWINDEntities nORTHWINDEntities = new EDM_WPF.NORTHWINDEntities();
            System.Windows.Data.CollectionViewSource ordersViewSource =
              ((System.Windows.Data.CollectionViewSource)
                    (this.FindResource("ordersViewSource")));
            ordersViewSource.Source = nORTHWINDEntities.Orders.Include("Order_Details");
        }
```

Test

Ein Test wird Sie von der Funktionsweise überzeugen:



Wie Sie sehen, können Sie bereits mit dem per Drag & Drop erzeugten Programm gut leben, einige Änderungen machen jedoch den Quellcode übersichtlicher und besser verständlich.

Kapitel 16

Druckausgabe mit WPF

Mit der Einführung von WPF wurde auch eine grundsätzlich andere Vorgehensweise bei der Druckausgabe von Dokumenten gewählt, die bei mehr als einer Druckausgabeseite schnell an Komplexität gewinnt. Erschwerend kommt hinzu, dass für Druckausgabe und Druckvorschau[1] keine einheitliche Lösung gewählt wurde, was es dem Programmierer auch nicht einfacher macht.

Aus diesem Grund wollen wir Sie zunächst mit der "Trivial-Lösung" für die Ausgabe einzelner Seiten sowie die Auswahl des Zieldruckers vertraut machen, bevor wir uns den komplexeren Themen wie

- Druckvorschau,
- mehrseitige Dokumente
- und Druckerauswahl/-konfiguration

zuwenden wollen. Doch bevor es so weit ist, wollen wir Sie in einer Kurzübersicht mit den neuen Konzepten der WPF-Druckausgabe vertraut machen.

16.1 Grundlagen

Eine Einführung zum Drucken in WPF wäre nicht komplett, wenn wir nicht kurz auf das Thema XPS-Dokumente eingehen würden.

16.1.1 XPS-Dokumente

Mit XPS (*XML Paper Specification*) versucht Microsoft einen Pendant zum derzeit weit verbreiteten Adobe PDF-Format zu etablieren. Im Grunde handelt es sich um eine geräteunabhängige vektororientierte Seitenbeschreibungssprache, die einen ungehinderten Informationsfluss aus der Anwendung bis zum finalen Ausgabegerät sicherstellen soll. Dafür findet sich nicht zuletzt auch ab Windows Vista ein entsprechender Druckertreiber, dessen Ausgaben in einer Datei landen, die Sie wiederum mit dem entsprechenden Betrachter anzeigen können.

[1] Eine direkte Druckvorschau-Komponente werden Sie auch nicht finden, wir nehmen dafür den *DocumentViewer*.

Über die Vor- und Nachteile zum bereits etablierten PDF lässt sicher sicher streiten, was aber nichts an der Tatsache ändert, dass XPS nun mal der zentrale Weg für die Druckausgaben unserer WPF-Anwendungen ist. Allerdings hat auch Microsoft schon richtig eingeschätzt, dass eine relevante XPS-Unterstützung für das neue Format von kaum einem Druckerhersteller zu erwarten ist, und so wird innerhalb der Druckausgabe eine Umwandlung der XPS-Daten in die bekannten GDI- bzw. PDL-Daten vorgenommen, um auch mit den derzeit am Markt befindlichen Geräten sinnvoll arbeiten zu können.

Um all diese Hintergründe müssen Sie sich als Programmierer jedoch nicht selbst kümmern, das für Sie interessante *XpsDocumentWriter*-Objekt nimmt, je nach Endgerät, die erforderliche XPS zu GDI-Umwandlung automatisch vor.

Für die Interaktion mit dem XPS-Ausgabesystem stehen Ihnen in C# die beiden Namespaces

- *System.Printing*
- und *System.Windows.Xps*

mit den entsprechenden Klassen zur Verfügung. Zusätzlich findet sich ganz unscheinbar im Namespace *System.Windows.Controls* auch eine *PrintDialog*-Komponente, die Sie in keinem Fall mit der entsprechenden Windows Forms-Komponente verwechseln sollten. Die *PrintDialog*-Komponente bietet neben der vermuteten Dialog-Funktionalität vor allem einen einfachen Zugriff auf die installierten Drucker, um zum Beispiel Controls (und das kann auch, wie von WPF gewohnt, eine geschachtelte Anordnung sein) oder auch ganze XPS-Dokumente zu drucken. Mehr dazu finden Sie im Abschnitt 16.2 ab Seite 841.

16.1.2 System.Printing

Dieser Namespace bietet mit den enthaltenen Klassen (Auszug)

- *PrintServer*
  (repräsentiert den aktuellen Druckserver, d h. den Computer)
- *PrintQueue*
  (repräsentiert den aktuellen Drucker und dessen Druckerwarteschlange)
- *PrintSystemJobInfo*
  (ein Druckjob und dessen Status)
- *PrintTicket*
  (die Konfiguration des Druckauftrags, wie Seitenformat, Seitendrehung etc.)

die Grundlage für die Arbeit mit der Druckausgabe in WPF.

HINWEIS: Bevor Sie auf diesen Namespace zugreifen können, müssen Sie unter Verweise noch die Assembly *System.Printing.dll* nachträglich einbinden. Je nach Funktionsumfang kann es auch nötig sein, zusätzlich die Assembly *ReachFramework.dll* einzubinden.

HINWEIS: Weitere Informationen zu den o.g. Klassen finden Sie im Abschnitt 16.4.

16.1.3 System.Windows.Xps

Wie schon erwähnt, ist XPS der Dreh- und Angelpunkt der WPF-Druckausgabe. Über den Namespace *System.Windows.Xps* stehen Ihnen die nötigen Schnittstellen-Klassen *VisualsToXpsDocument* und *XpsDocumentWriter* zur Verfügung.

Zusätzlich finden sich weitere Namespaces wie *System.IO.Packaging* und *System.Windows.Xps.Packaging*, die im Zusammenhang mit der Ausgabe von XPS-Dateien eine Rollen spielen. Hier auf alle Einzelheiten einzugehen, würde den Rahmen des Kapitels sprengen.

HINWEIS: Bevor Sie auf diesen Namespace komplett zugreifen können, müssen Sie Verweise auf die Assemblies *System.Printing.dll* und *ReachFramework.dll* einbinden.

16.2 Einfache Druckausgaben mit dem PrintDialog

Hoffentlich haben wir Sie mit den Kurzausführungen zu XPS nicht ganz verschreckt, nicht in jedem Fall müssen Sie sich mit der Gesamtkomplexität der WPF-Druckausgabe herumschlagen, es geht teilweise auch ganz einfach, wie es auch das folgende Beispiel zeigt:

BEISPIEL 16.1: Verwendung von *PrintDialog*

C#

```
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
```

Instanz erstellen:

```
            PrintDialog dlg = new PrintDialog();
```

Dialog anzeigen:

```
            if (dlg.ShowDialog() == true)
            {
```

Wird auf die Ok-Taste geklickt, drucken wir ein Control aus:

```
                dlg.PrintVisual(Button2, "Erster Test");
            }
        }
```

Ergebnis

Das Ergebnis dürfte etwa so aussehen:

Der Druckdialog selbst wird Ihnen sicher von den Windows Forms her noch bekannt sein, die Funktionalität ist zunächst gleich. Doch nach der Auswahl von Drucker, Seitenbereich etc. geht es hier erst richtig los. Über die Methode *PrintVisual* besteht die Möglichkeit, XAML-Elemente Ihres Formulars (oder auch gleich das ganze Formular) direkt an den Drucker zu senden. Da diese ohnehin vektorbasiert sind, ist auch die Druckqualität im Vergleich zu Windows Forms wesentlich besser.

HINWEIS: Wer jetzt befürchtet, immer den Dialog anzeigen zu müssen, liegt falsch, lassen Sie den Aufruf der Methode *ShowDialog* weg, wird automatisch der Standard-Drucker verwendet.

Doch Vorsicht – der oben gezeigte Aufruf von *PrintVisual* hat einige Einschränkungen, die Sie kennen sollten, bevor Sie darauf basierend Ihre Programme umschreiben:

1. Es wird immer nur **eine** Seite bedruckt, ist der zu druckende Content größer, wird er abgeschnitten.
2. Der Ausdruck wird immer an der linken oberen Blattecke ausgerichtet. Da viele Drucker jedoch einen Seitenoffset besitzen, wird in diesem Fall meist etwas am oberen und linken Rand abgeschnitten.

Als Lösung für 1. bietet sich die komplexere *PrintDocument*-Methode an, auf die wir noch zurückkommen werden, Problem 2 können Sie umgehen, wenn Sie dem zu druckenden Control einen entsprechenden *Margin* verpassen, der automatisch mit den Seitenrändern verrechnet wird.

Wo wir schon bei Seitenrändern sind: Über den *PrintDialog* können Sie auch einige wichtige Informationen zum aktuellen Ausgabemedium[1] in Erfahrung bringen:

- Seitenhöhe und Breite (*PrintTicket.PageMediaSize.Height, PrintTicket.PageMediaSize.Width*)
- Druckbereichshöhe und -breite (*PrintableAreaHeight, PrintableAreaWidth*)
- Randloser Druck möglich (*PrintTicket.PageBorderless*)
- Seitenausrichtung (*PrintTicket.PageOrientation*)

HINWEIS: Achtung: Ist das Blatt gedreht, hat dies keinen Einfluss auf die Seiten- bzw. Druckbereichsabmessungen, d h., die Papierbreite ist beim Querformat mit *PrintableAreaHeight* zu bestimmen.

Natürlich gibt es noch dutzende weitere Eigenschaften, in den folgenden Abschnitten kommen wir noch darauf zurück.

Doch was ist eigentlich mit reiner Textausgabe? Hier sollten Sie den Namen der *PrintVisual*-Methode nicht zu genau nehmen, mit dieser Methode können Sie natürlich auch Text bzw. ein *TextBlock*-Objekt ausgeben und das auf wesentlich komfortablere Art als bei den Windows Forms. Der *TextBlock* darf natürlich auch über Formatierungen etc. verfügen (siehe Seite 618).

[1] tolle Umschreibung für "Blatt"

BEISPIEL 16.2: Textausgabe auf dem Drucker

C#

```
        private void Button2_Click(object sender, RoutedEventArgs e)
        {
            PrintDialog dlg = new PrintDialog();
            if (dlg.ShowDialog() == true)
            {
```

Neue *TextBlock*-Instanz unabhängig vom Formular erzeugen:

```
                TextBlock txt = new TextBlock();
```

Ränder festlegen, so umgehen wir das Problem mit dem Seitenoffset:

```
                txt.Margin = new Thickness(15);
```

Hier der eigentliche Ausgabetext:

```
                txt.Text = "Hier folgt ein selten belangloser Text, den Sie besser nicht
lesen sollten. Aber so wird Ihnen schnell klar, wie einfach die Textausgabe ist.";
```

Schriftgröße und Schriftart bestimmen:

```
                txt.FontSize = 25;
                txt.FontFamily = new FontFamily("Arial");
```

Textumbruch aktivieren:

```
                txt.TextWrapping = TextWrapping.Wrap;
```

Und jetzt müssen wir uns um die Größe des Controls kümmern (Layout-Aktualisierung), sonst ist es nicht zu sehen:

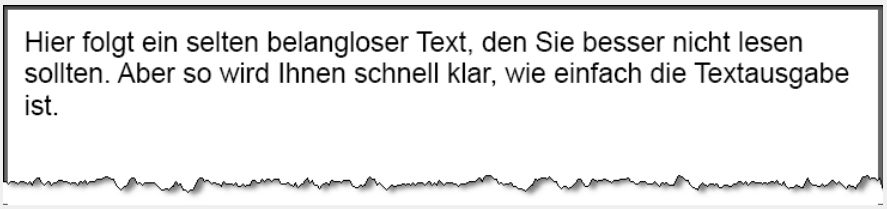
```
                txt.Measure(new Size(dlg.PrintableAreaWidth, dlg.PrintableAreaHeight));
                txt.Arrange(new Rect(0, 0, txt.DesiredSize.Width, txt.DesiredSize.Height));
```

Last, but no least, die Druckausgabe:

```
                dlg.PrintVisual(txt, "Erster Test");
            }
        }
```

Ergebnis

Damit kann man doch schon ganz zufrieden sein:



Damit wollen wir uns aus der "Programmierer-Kuschelecke" verabschieden und uns in die "Wildnis" von Druckvorschau und mehrseitiger Ausgabe wagen.

16.3 Mehrseitige Druckvorschau-Funktion

Puh..., ganz schön viele Wünsche auf einmal und WPF lässt uns an dieser Stelle doch etwas im Regen stehen. Ein Blick in den Werkzeugkasten verheißt nichts Gutes, weder für das eine noch für das andere findet sich **die** Standardlösung.

Wer den Blick in die Hilfe bzw. ins Internet wagt, findet sicher auch recht schnell die Standardlösung mit einer abgeleitetem *DocumentPaginator*-Klasse, deren Methoden Sie implementieren müssen. Doch so ganz befriedigend ist diese Lösung nicht, von der Übersicht ganz zu schweigen.

Aus diesem Grund haben sich die Autoren diverse Einzellösungen im Internet angesehen und aus all diesen Hinweisen/Lösungsvorschlägen etc. zwei halbwegs nutzbare Lösungen realisiert, die sowohl unsere Wünsche nach mehrseitigen Dokumenten erfüllen als auch eine Druckvorschau bieten.

Doch wie immer findet sich auch hier ein Haar in der Suppe: Sie müssen sich entscheiden, ob Sie mit einem flexiblen Dokument arbeiten, das über Fließtext mit eingebetteten Grafiken etc. verfügt (ein typisches Flow-Dokument) oder ob Sie die einzelnen Elemente Layout-orientiert bzw. absolut anordnen wollen (Fix-Dokument). Für beides finden Sie im Folgenden die Lösung.

16.3.1 Fix-Dokumente

Sehen wir uns zunächst die layout- und seitenorientierte Ausgabe von Dokumenten an. Wie auch bei den noch zu betrachtenden Flow-Dokumenten wollen wir die erforderliche Funktionalität in einer eigenen Klasse kapseln, um uns ganz auf die reine Ausgabe konzentrieren zu können. Ziel war es, mit so wenig Code wie möglich eine Druckausgabe und eine Druckvorschaufunktion zu implementieren.

Einsatzbeispiel

Bevor wir uns in die Details vertiefen, wollen wir uns ansehen, wie wir die spätere Klasse *FixedPrintManager* einsetzen können und was wir alles zu Papier bringen können. Sie werden sehen, dass viele Konzepte bereits in den vorhergehenden Kapiteln erläutert wurden.

BEISPIEL 16.3: Verwendung der Klasse *FixedPrintManager*

C#

Binden Sie zunächst die beiden Assemblies *System.Printing.dll* und *ReachFramework.dll* ein. Erstellen Sie nachfolgend ein Formular, in das Sie eine Schaltfläche und einen *DocumentViewer*[1] einbinden.

Zunächst die nötigen Namespaces:

```
...
using System.IO;
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
using System.Windows.Markup;
```

[1] Sie ahnen es sicher schon, der *DocumentViewer* wird unsere Druckvorschau.

BEISPIEL 16.3: Verwendung der Klasse *FixedPrintManager*

```
using System.IO.Packaging;
...
```

Nach dem Klick auf die Schaltfläche setzen die hektischen Aktivitäten ein:

```
private void Button1_Click(object sender, RoutedEventArgs e)
{
```

Als Erstes definieren wir einige Objekte, die wir später zu Papier bringen wollen (nähere Informationen dazu finden Sie in den vorhergehenden Kapiteln.):

```
TextBlock txt;
Image img;
StackPanel panel;
```

Wir erstellen eine Instanz unserer *FixedPrintManager*-Klasse:

```
FixedPrintManager pm = new FixedPrintManager(
                        borders: new Thickness(15), ShowPrintDialog: true);
```

Parameter sind die Breite der Seitenränder und die Option, ob ein Druckerauswahldialog angezeigt werden soll.

Und damit sind wir auch schon beim Darstellen der ersten Seite:

Zentrales Element ist ein *StackPanel*, wie Sie es auch von den WPF-Formularen kennen:

```
panel = new StackPanel();
```

In das *StackPanel* fügen wir einen *TextBlock* ein. Die entsprechende Instanz erzeugen wir allerdings per *NewTextBlock*-Methode unserer *FixedPrintManager*-Instanz:

```
txt = pm.NewTextBlock("Times New Roman", 40);
```

Der Vorteil: Wir müssen uns nicht um das recht umständliche Konfigurieren des *TextBlock*s kümmern, Defaultparameter ermöglichen das einfache und komfortable Setzen der Eigenschaften.

Noch etwas Text definieren und den *TextBlock* dem *StackPanel* hinzufügen:

```
txt.Text =
  "Hier steht zum Beispiel jede Menge Text. Hier steht zum Beispiel
     jede Menge Text.Hier steht zum Beispiel jede Menge Text. ..... ";
panel.Children.Add(txt);
```

Ein weiterer *TextBlock:*

```
txt = pm.NewTextBlock(text: "Hier steht zum Beispiel jede Menge Text.
      Hier steht zum Beispiel jede Menge Text.Hier steht zum .....");
txt.Te xtAlignment = TextAlignment.Justify;
txt.Margin = new Thickness(0, 10, 0, 20);
panel.Children.Add(txt);
```

BEISPIEL 16.3: Verwendung der Klasse *FixedPrintManager*

Ein *Image* in das *StackPanel* einfügen:

```
img = new Image();
```

Im Gegensatz zu den beiden vorhergehenden Beispielen müssen wir im Fall des *Image*-Controls selbst für die Größenanpassung des *Image* sorgen:

```
img.Width = pm.PageSize.Width - pm.Borders.Left - pm.Borders.Right;
img.Stretch = Stretch.Uniform;
img.Source = new BitmapImage(new Uri("pack://application:,,,/Desert.jpg"));
panel.Children.Add(img);
```

Wir schließen die erste "Druck"-Seite ab, indem wir das Stackpanel an die *NewPage*-Methode übergeben:

```
pm.NewPage(panel);
```

Hier erzeugen wir eine reine Textseite:

```
txt = pm.NewTextBlock(Fontsize: 22);
txt.Text = "Hier ist die zweite Seite mit Fließtext. Hier ist die zweite
           Seite mit Fließtext. Hier ist die zweite Seite ..... ";
```

Abschließen der zweiten Seite:

```
pm.NewPage(txt);
```

Ein ganz triviales Beispiel für die Ausgabe einzelner Controls:

```
pm.NewPage(new Calendar());
```

Natürlich können Sie bei der Druckausgabe auch alle Bildtransformationen nutzen, die Sie bereits kennen gelernt haben.

Als Grundlage dient uns in diesem Fall ein *Canvas,* der auch das absolute Positionieren des Controls zulässt:

```
Canvas cv = new Canvas();
img = new Image();
img.Width = pm.PageSize.Width - pm.Borders.Left - pm.Borders.Right;
img.Stretch = Stretch.Uniform;
img.Source = new BitmapImage(new Uri("pack://application:,,,/Desert.jpg"));
img.SetValue(Canvas.TopProperty, 100.0);
img.SetValue(Canvas.LeftProperty, 1.0);
```

Die Transformation anwenden:

```
img.LayoutTransform = new RotateTransform(45);
cv.Children.Add(img);
pm.NewPage(cv);
```

Abschließend möchten wir Ihnen an einer weiteren "Druckseite" demonstrieren, wie Sie auch XAML-Code aus den Ressourcen des aktuellen Formulars für die Druckausgabe verwenden können:

BEISPIEL 16.3: Verwendung der Klasse *FixedPrintManager*

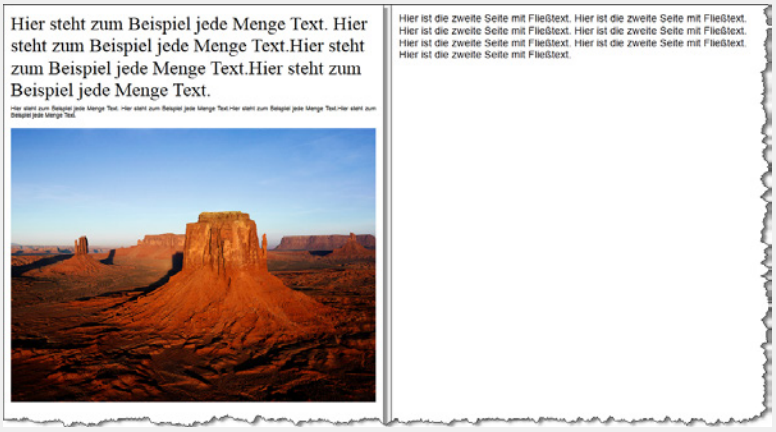
```
            cv = (Canvas)Resources["ResourceData"];
            cv.Measure(pm.PageSize);
            cv.Arrange(new Rect(0, 0, cv.DesiredSize.Width, cv.DesiredSize.Height));
            pm.NewPage(cv);
```

So, damit ist das Dokument erstellt, Sie haben jetzt zwei Möglichkeiten: Entweder Sie zeigen das Dokument in einem *DocumentViewer* an oder Sie gehen sofort zum Druck über:

```
            documentViewer1.Document = pm.Document;
            pm.Print();
        }
```

Ergebnis

Das Ergebnis unserer Bemühungen zeigen die folgenden Abbildungen:



Vielleicht sind Sie auch der Meinung, dass der Aufwand für das Erstellen des Dokuments nicht allzu hoch war. Die Verwendung der Textblöcke hatte sogar den Vorteil, dass wir uns um Zeilenumbrüche keinen Kopf machen mussten. Die restlichen Ausgaben waren weitgehend mit der Formulardarstellung identisch, neue Konzepte müssen Sie bei dieser Form der Druckausgabe nicht erlernen.

Die Klasse FixedPrintManager

Doch nun wollen wir einen Blick auf die verwendete Klasse *FixedPrintManager* werfen:

```
public class FixedPrintManager
{
```

Einige interne Variablen:

```
    private FixedDocument _PrintDocument;
    private Size _PageSize;
    private Thickness _Borders;
    private PrintDialog _dlg;
```

Die *Borders*-Eigenschaft (Abfrage der Seitenränder):

```
    public Thickness Borders
    { get { return _Borders; }  }
```

Sie *PageSize*-Eigenschaft (Abfrage der Seitengröße, diese wird im Konstruktor gesetzt):

```
    public Size PageSize
    { get { return _PageSize; } }
```

Die Rückgabe des Dokuments für die Druckvorschau:

```
    public FixedDocument Document
    { get { return _PrintDocument; } }
```

Der Konstruktor unserer Klasse:

```
    public FixedPrintManager(Thickness borders, bool ShowPrintDialog = false)
    {
```

Ein neues *FixedDocument* erstellen:

```
        _PrintDocument = new FixedDocument();
```

Intern nutzen wir den *PrintDialog* für die Druckausgabe und die Bestimmung von Drucker und Seitenrändern:

```
        _dlg = new PrintDialog();
        if (ShowPrintDialog) _dlg.ShowDialog();
        _PageSize = new Size(_dlg.PrintableAreaWidth, _dlg.PrintableAreaHeight);
        _PrintDocument.DocumentPaginator.PageSize = _PageSize;
        _Borders = borders;
    }
```

Das Starten der Druckausgabe erfolgt mit der folgenden Methode:

```
public void Print(string title = "Mein Druckauftrag")
{
    _dlg.PrintDocument(_PrintDocument.DocumentPaginator, title);
}
```

Zum Erstellen einer neuen Seite nutzen Sie folgende Methode:

```
public void NewPage(UIElement content)
{
```

Erzeugen einer neuen *FixedPage,* diese wird im Folgenden in das interne *FixedDocument* eingefügt:

```
    FixedPage _page = new FixedPage();
```

Seitengröße und -ränder bestimmen:

```
    _page.Width = _PrintDocument.DocumentPaginator.PageSize.Width;
    _page.Height = _PrintDocument.DocumentPaginator.PageSize.Height;
    _page.Margin = _Borders;
```

Den übergeben Seiteninhalt einfügen:

```
    _page.Children.Add(content);
```

Hier wir es etwas komplizierter, da ein direkter Zugriff auf die *AddChild*-Methode nicht möglich ist:

```
    PageContent _pageContent = new PageContent();
    ((IAddChild)_pageContent).AddChild(_page);
```

Anhängen an das *FixedDocument*:

```
    _PrintDocument.Pages.Add(_pageContent);
}
```

Last but not least, noch unsere Methode zum Erstellen von Textblöcken. Diese macht ausgiebig von der Verwendung optionaler Parameter Gebrauch und nimmt uns das lästige Parametrieren des *TextBlock*s ab:

```
public TextBlock NewTextBlock(string Fontname = "Arial", int Fontsize = 12,
                              string text = "")
{
    TextBlock txt = new TextBlock();
    txt.Width = _PageSize.Width - _Borders.Left - _Borders.Right;
    txt.FontFamily = new FontFamily(Fontname);
    txt.FontSize = Fontsize;
    txt.Text = text;
    txt.TextWrapping = TextWrapping.WrapWithOverflow;
    return txt;
}
}
```

Damit haben Sie bereits ein recht umfangreiches Grundgerüst für eigene Erweiterungen. Insbesondere die Ausgabe von Linien-Zeichnungen etc. ist sicher noch verbesserungswürdig, aber Sie wollen ja auch noch etwas zu tun haben.

16.3.2 Flow-Dokumente

Nachdem wir uns mit der seitenorientierten Methode, d h. den *FixedDocuments* beschäftigt haben, wollen wir und mit den flexibleren Flow-Dokumenten befassen (siehe auch 13.18). Deren Vorteil liegt in der Beschreibung von formatierten Fließtexten, die wiederum aus Absätzen (*Paragraph*), eingefügten Controls (*BlockUIContainer*), Auflistungen (*List*), *Section*s und Tabellen (*Table*) bestehen können. Um Seitenränder, Seitengrößen und -ausrichtungen müssen Sie sich zum Zeitpunkt der Dokumenterstellung keinen Kopf machen, dies erfolgt automatisch bei der finalen Ausgabe auf dem jeweiligen Ausgabegerät.

Einführungsbeispiel

Ein kleines Beispiel soll Sie von den Vorzügen unserer *FlowPrintManager*-Klasse überzeugen, die wie auch die *FixedDocument*-Klasse über eine *Document*-Eigenschaft (für die Druckvorschau) und eine *Print*-Methode verfügt.

BEISPIEL 16.4: Verwendung der *FlowPrintManager*-Klasse

C#

Auch hier binden Sie zunächst die beiden Assemblies *System.Printing.dll* und *ReachFramework.dll* ein. Erstellen Sie nachfolgend ein Formular, in das Sie eine Schaltfläche und einen *DocumentViewer*[1] einbinden.

```
private void Button2_Click(object sender, RoutedEventArgs e)
{
```

Eine Instanz unserer Klasse erstellen:

```
FlowPrintManager fpm = new FlowPrintManager(new Thickness(75, 25,25,25), true);
```

Wir erzeugen einen neuen Abschnitt:

```
fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum Beispiel jede Menge
Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum Beispiel jede Menge Text.Hier
steht zum Beispiel jede Menge Text.")));
```

Einen leeren Absatz:

```
fpm.FlowDoc.Blocks.Add(new Paragraph(new LineBreak()));
      for (int i=0; i<40; i++)
```

Und hier noch ein paar weitere Absätze:

```
for (int i=0; i<40; i++)
   fpm.FlowDoc.Blocks.Add(new Paragraph(new Run("Hier steht zum Beispiel jede
Menge Text. Hier steht zum Beispiel jede Menge Text.Hier steht zum Beispiel jede Menge
```

[1] Sie ahnen es sicher schon, der *DocumentViewer* wird unsere Druckvorschau.

BEISPIEL 16.4: Verwendung der *FlowPrintManager*-Klasse

```
Text.Hier steht zum Beispiel jede Menge Text.")));
```

Uns schon können Sie dieses Dokument in der Druckvorschau anzeigen:

```
            documentViewer1.Document = fpm.Document;
        }
```

Ergebnis

Die Druckvorschau bringt es an den Tag, durch unsere kleine Schleife bei der Ausgabe des letzten Absatzes wird der Text so lang, dass er nicht mehr auf eine Seite passt:

Doch wie funktioniert die Umwandlung des oben erzeugten *FlowDocument* in ein *FixedDocument*, das wir für die Druckvorschau (*DocumentViewer*) benötigen?

Die Klasse FlowPrintManager

Hier hilft uns die Möglichkeit weiter, mittels *XpsDocumentWriter* das *FlowDocument* in ein XPS-Dokument zu schreiben und aus diesem wiederum ein *FixedDocument* zu erzeugen. Doch leider ist diese Lösung im Normalfall mit einer physischen Datei verbunden, was nicht nur unschön aussieht, sondern teilweise auch Probleme nach sich ziehen kann. Aus diesem Grund haben wir uns für den Weg über einen *MemoryStream* entschieden, das komplette Handling erfolgt also im Speicher.

Doch nun zu den Details unserer *FlowPrintManager*-Klasse:

```
...
    public class FlowPrintManager
    {
```

Die internen Variablen:

```
private FlowDocument _flowdocument;
private FixedDocumentSequence _document;
private MemoryStream ms;
private Package package;
private PrintDialog _dlg;
```

Die Eigenschaft *FlowDoc*, über die wir unsere *FlowDocument* zusammenbasteln können:

```
public FlowDocument FlowDoc
{
    get {return _flowdocument; }
}
```

Die Eigenschaft *Document* liefert uns die gewünschte *FixedDocumentSequence* für die Druckvorschau:

```
public FixedDocumentSequence Document
{
    get
    {
```

Und hier wird es schnell etwas unübersichtlich. Grundlage des Schreibens von XPS-Dokumenten ist zunächst ein Package (ein Container mit Kompression), in welches das eigentliche XPS-Dokument eingefügt wird.

Am Anfang löschen wir zunächst einmal pauschal ein möglicherweise vom letzten Durchlauf noch vorhandenes Package:

```
PackageStore.RemovePackage(new Uri("memorystream://data.xps"));
```

Neues Package erzeugen:

```
PackageStore.AddPackage(new Uri("memorystream://data.xps"), package);
```

Neues *XpsDocument* im Package erzeugen:

```
XpsDocument xpsDocument = new XpsDocument(package, CompressionOption.Fast,
                          "memorystream://data.xps");
```

XpsDocumentWriter für das *XpsDocument* erstellen:

```
XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(xpsDocument);
```

Schreiben der Daten per *XpsDocumentWriter*:

```
writer.Write(((IDocumentPaginatorSource)_flowdocument).DocumentPaginator);
```

Für das neue XPS-Dokument rufen wir eine *FixedDocumentSequence* ab und geben diese zurück:

```
            _document = xpsDocument.GetFixedDocumentSequence();
            xpsDocument.Close();
            return _document;
        }
    }
```

Unser Konstruktor kümmert sich um das Initialisieren der internen Variablen:

```
        public FlowPrintManager(Thickness borders, bool ShowPrintDialog = false)
        {
```

MemoryStream und *Package* erzeugen:

```
            ms = new MemoryStream();
            package = Package.Open(ms, FileMode.Create, FileAccess.ReadWrite);
```

Eventuell den *PrintDialog* anzeigen:

```
            _dlg = new PrintDialog();
            if (ShowPrintDialog) _dlg.ShowDialog();
```

Standardeinstellungen vornehmen:

```
            _flowdocument = new FlowDocument();
            _flowdocument.ColumnWidth = _dlg.PrintableAreaWidth;
            _flowdocument.PageHeight = _dlg.PrintableAreaHeight;
            _flowdocument.PagePadding = borders;
        }
```

Für die direkte Druckausgabe nutzen wir wieder den *PrintDialog*:

```
        public void Print(string title = "Mein Druckauftrag")
        {
            _dlg.PrintDocument(this.Document.DocumentPaginator, title);
        }
    }
```

Damit haben Sie auch eine recht einfache Lösung für die Ausgabe von Flow-Dokumenten.

Im folgenden Abschnitt steht nach der bisherigen Arbeit mit dem *PrintDialog*-Control ein intensiverer Blick auf die Druckerdetails und die Druckerkonfiguration im Mittelpunkt.

16.4 Druckerinfos, -auswahl, -konfiguration

Haben Sie bereits mit dem *PrintDialog*-Control gearbeitet, hat Sie sicher auch gestört, dass zur Druckerauswahl ein Dialog angezeigt wird, der gleichzeitig die Anzahl der zu druckenden Seiten etc. bestimmt. Das ist nicht in jedem Fall gewünscht, viel besser ist der direkte Zugriff auf die installierten Drucker und deren Eigenschaften.

An dieser Stelle müssen Sie sich von Ihren bisherigen Kenntnissen der Druckausgabe (Windows Forms) verabschieden, in WPF haben Sie es mit gänzlich anderen Klassen und Eigenschaften zu tun.

HINWEIS: Für die weitere Arbeit mit den entsprechenden Klassen müssen Sie die Assembly *System.Printing.dll* in ihr WPF-Projekt einbinden.

16.4.1 Die installierten Drucker bestimmen

Ausgangspunkt für die Bestimmung der installierten Drucker ist die *LocalPrintServer*-Klasse, die den aktuellen Computer mit seinen angeschlossenen Druckern (inklusive PDF-, XPS-Drucker etc.) kapselt.

HINWEIS: Mit *PrintServer*-Klasse und den entsprechenden Konstruktoren können Sie auch auf externe Printserver zugreifen. Wir beschränken uns im Weiteren jedoch auf den aktuellen PC.

Nach dem Erstellen einer Instanz können Sie über die *GetPrintQueues*-Methode eine Liste der installierten Drucker (hier per *PrintQueue* gekapselt) abrufen.

BEISPIEL 16.5: Anzeige einer Liste aller Drucker

C#

```
...
using System.Printing;
using System.Collections;
...
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
```

Den lokalen *PrintServer* abrufen:

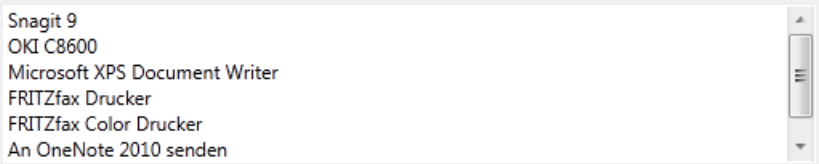
```
            var server = new LocalPrintServer();
```

Alle Drucker bestimmen und in einer *ListBox* anzeigen:

```
            var queues = server.GetPrintQueues();
            ListBox1.ItemsSource = queues;
            ListBox1.DisplayMemberPath = "Name";
        }
```

Ergebnis

Die angezeigte Liste:



Da der *ListBox* die *PrintQueue*-Objekte zugewiesen wurden, ist es nachfolgend auch recht einfach, auf die Details des jeweiligen Druckers zuzugreifen (siehe 16.4.3).

16.4.2 Den Standarddrucker bestimmen

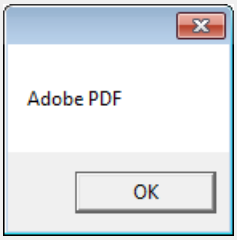
Erstellen Sie eine Instanz der *LocalPrintServer*-Klasse, können Sie über die Eigenschaft *DefaultPrintQueue* den Standarddrucker und damit dessen Eigenschaften bestimmen.

BEISPIEL 16.6: Standarddrucker bestimmen

C#

```
...
            var server = new LocalPrintServer();
            var queue = server.DefaultPrintQueue;
            MessageBox.Show(queue.FullName);
...
```

Ergebnis



16.4.3 Mehr über einzelne Drucker erfahren

Damit sind wir bereits bei der *PrintQueue*-Klasse angekommen. Diese ist recht auskunftsfreudig und bietet eine reiche Palette an Eigenschaften, mit denen Sie mehr über einen spezifischen Drucker erfahren können. Wie Sie den aktuellen Standarddrucker bzw. dessen zugehöriges *PrintQueue*-Objekt bestimmen haben Sie bereits im vorhergehenden Abschnitt erfahren. Um an ein *PrintQueue*-Objekt aus der *ListBox* (siehe Seite 854 "Die installierten Drucker bestimmen") zu gelangen, können Sie beispielsweise das *SelectionChanged*-Ereignis nutzen:

```
private void ListBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    var queue = ((PrintQueue)ListBox1.SelectedItem);
```

Die auf den ersten Blick unscheinbare *PrintQueue*-Klasse entpuppt sich bei genauerem Hinsehen jedoch als ergiebige Informationsquelle. Die folgende Tabelle zeigt eine Übersicht der wichtigsten Eigenschaften:

Eigenschaft	Beschreibung
AveragePagesPerMinute	Druckgeschwindigkeit in Seiten/Minute
Comment	Kommentar zum Drucker (lesen/schreiben)
CurrentJobSettings	Druckeinstellungen für den aktuellen Druckjob
DefaultPrintTicket	Die Standarddruckeinstellungen des Druckers
DefaultPriority	Die Priorität für neue Druckjobs
Description	Beschreibung für den Drucker

Eigenschaft	Beschreibung
FullName	Name des Druckers
HasPaperProblem	Gibt es Papierprobleme?
HasToner	Gibt es Tonerprobleme?
HostingPrintServer	Printserver, der den Druckjob kontrolliert
IsBidiEnabled	Ist bidirektionale Kommunikation mit dem Drucker möglich?
IsBusy	Ist der Drucker beschäftigt?
IsDirect	Wird direkt gedruckt oder gespoolt?
IsManualFeedRequired	Manueller Papiervorschub nötig?
IsNotAvailable	Ist der Drucker verfügbar?
IsOffline	Ist der Drucker offline?
IsOutOfMemory	Ist zu wenig Speicher verfügbar?
IsOutOfPaper	Ist das Papier alle?
IsOutputBinFull	Ist das Ausgabefach voll?
IsPaperJammed	Hat sich Papier im Drucker verfangen?
IsPaused	Pausiert die Druckerwarteschlange?
IsPendingDeletion	Wird ein Druckjob gerade gelöscht?
IsPowerSaveOn	Ist der Drucker im Energiesparmodus?
IsPrinting	Druckt der Drucker gerade?
IsProcessing	Wird ein Druckjob abgearbeitet?
IsPublished	Ist der Drucker für andere Nutzer sichtbar (Netzwerk)?
IsQueued	Wird eine Druckerwarteschlange angeboten?
IsRawOnlyEnabled	Können EMF (Enhanced Meta File) Daten zum Drucken verwendet werden?
IsServerUnknown	Ist der Drucker in einem Fehlerzustand?
IsShared	Ist der Drucker für andere Nutzer freigegeben (Netzwerk)?
IsTonerLow	Ist der Toner alle?
IsWaiting	Wartet der Drucker auf einen Druckjob?
IsWarmingUp	Ist der Drucker in der Warmlaufphase?
IsXpsDevice	Unterstützt der Drucker XML Paper Specification (XPS) als Seitenbeschreibungssprache?
KeepPrintedJobs	Wird der Druckjob abgespeichert?
Location	Wo steht der Drucker?
Name	Der Druckername
NeedUserIntervention	Muss ein Anwender den Drucker bedienen?
NumberOfJobs	Die Anzahl der Druckjobs
PagePunt	Warum kann die aktuelle Seite nicht gedruckt werden?
PrintingIsCancelled	Wurde der Druckjob abgebrochen?

Eigenschaft	Beschreibung
PropertiesCollection	Eine Collection von Attribut/Wert-Paaren
QueueAttributes	Druckerqueue-Eigenschaften
QueueDriver	Der verwendete Druckertreiber
QueuePort	Der Druckerport
QueuePrintProcessor	Der verwendete Druckprozessor
QueueStatus	Queue-Status (z.B. "warming up" "initializing" "printing")
SeparatorFile	Datei mit dem Deckblatt für einzelne Druckjobs
ShareName	Freigabename
StartTimeOfDay	Wann wurde der erste Druckjob gestartet?
UntilTimeOfDay	Wann wurde der letzte Druckjob gestartet?
UserPrintTicket	Die gewünschten Druckeinstellungen des Anwenders

Ups, so viele Informationen über den Drucker und den Druckspooler hatten Sie bisher sicher nur mit intensiver Hilfe der API in Erfahrung bringen können. Da hat sich seit den Windows Forms doch einiges gebessert. Ganz nebenbei können Sie mit den Methoden *Pause*, *Resume* und *Purge* der *PrintQueue*-Klasse auch den Druckspooler steuern bzw. Druckaufträge anhalten, fortsetzen und löschen. Doch an dieser Stelle verweisen wir Sie dann besser an die Hilfe zu *PrintQueue-Klasse*.

HINWEIS: Wichtigste Methode der *PrintQueue*-Klasse ist die *CreateXpsDocumentWriter*-Methode, mit der Sie den für die Druckausgabe erforderlichen *XpsDocumentWriter* erhalten (siehe Seite 859).

16.4.4 Spezifische Druckeinstellungen vornehmen

Beim Betrachten der vorhergehenden Tabelle ist Ihnen sicher auch aufgefallen, dass Druckjob-spezifische Eigenschaften bisher nicht aufgetaucht sind. Einstellungen zum Druckjob selbst nehmen Sie über die *UserPrintTicket*-Eigenschaft vor, welche die per *DefaultPrintTicket* vorgegebenen Standard-Einstellungen überschreibt.

Eigenschaften von *UserPrintTicket*:

Name	Beschreibung
Collation	Sortierfolge bei Druck von Kopien
CopyCount	Anzahl der Kopien
DeviceFontSubstitution	Verwendung von Druckerschriftarten
Duplexing	Doppelseitiger Druck
InputBin	Welcher Einzug soll verwendet werden?
OutputColor	Wie werden Farbe/Graustufen behandelt?
OutputQuality	Druckqualität

Name	Beschreibung
PageBorderless	Kann randlos gedruckt werden?
PageMediaSize	Papiergröße
PageMediaType	Papierart
PageOrder	Druckreihenfolge (1.. n oder n ... 1)
PageOrientation	Seitenausrichtung
PageResolution	Seitenauflösung
PageScalingFactor	Skalierfaktor
PagesPerSheet	Druckseiten pro Blatt
PagesPerSheetDirection	Wie werden mehrere Seiten pro Blatt angeordnet?
PhotoPrintingIntent	Bei Unterstützung für Photodruck können hier verschiedene Qualitäten ausgewählt werden
Stapling	Optionen für das automatische Heften
TrueTypeFontMode	Optionen für die Verwendung von True Type Fonts (TTF).

Ein kleines Beispiel für die Konfiguration gefällig:

BEISPIEL 16.7: Druckausgabe umstellen auf Querformat mit zwei Kopien

C#
```
            var server = new LocalPrintServer();
            var queue = server.DefaultPrintQueue;
            queue.UserPrintTicket.PageOrientation = PageOrientation.Landscape;
            queue.UserPrintTicket.CopyCount = 2;
...
```

Auch hier finden Sie mittlerweile Einstellungen, für die Sie vor nicht allzu langer Zeit noch mit dem Druckertreiber intensiven "Gedankenaustausch" betreiben konnten.

Doch sicher stellt sich manchem auch die Frage, welche Werte muss/kann man überhaupt an die einzelnen Eigenschaften übergeben, bzw. was bedeuten die Rückgabewerte dieser Eigenschaften (was bedeutet beispielsweise *OutputColor = 2*).

Hier hilft Ihnen die *GetPrintCapabilities*-Methode der *PrintQueue*-Klasse weiter. Diese gibt über Ihre Eigenschaften (diese entsprechen weitgehend der vorhergehenden Tabelle) jeweils Collections mit der Bedeutung der Einträge zurück.

Statt vieler Worte ist sicher ein Beispiel recht hilfreich:

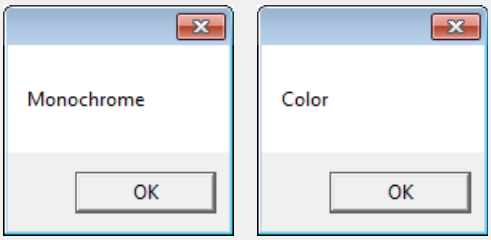
BEISPIEL 16.8: Welche Farbausgabeoptionen gibt es

C#
```
            var server = new LocalPrintServer();
            var queue = server.DefaultPrintQueue;
            var caps = queue.GetPrintCapabilities();
            foreach (var cc in caps.OutputColorCapability)
                MessageBox.Show(cc.ToString());
```

BEISPIEL 16.8: Welche Farbausgabeoptionen gibt es

Ergebnis

Die Anzeige:



Da es sich um eine normale Collection handelt, wissen Sie jetzt, dass 0 der "Monochromen"- und 1 der "Color"-Wiedergabe entspricht. Sie können an die *UserPrintTicket.OutputColor*-Eigenschaft also 0 oder 1 mit obiger Bedeutung übergeben. Alternativ lässt sich so auch ein entsprechender Wert richtig deuten.

HINWEIS: Obige Methode dürfte also für die Gestaltung von eigenen Druckdialogen von existenzieller Wichtigkeit sein, können Sie doch hier gleich die Werte für mögliche Auswahllisten abfragen.

Doch last, but not least, wollen wir natürlich etwas zu Papier bringen, womit wir auch schon beim letzten Absatz zu diesem Thema angekommen sind.

16.4.5 Direkte Druckausgabe

Nachdem Sie den Drucker ausreichend abgefragt und konfiguriert haben, soll natürlich auch ein Ergebnis auf dem Papier erscheinen. Hier schließt sich dann der Kreis zu den in den vorherigen Abschnitten vorgestellten Druckmöglichkeiten. Über ein per *CreateXpsDocumentWriter*-Methode erzeugtes *XpsDocumentWriter*-Objekt gelangen Ihre Ausgaben direkt an den gewünschten Drucker bzw. dessen Spooler.

BEISPIEL 16.9: Druckausgabe einer Textseite

C#

Standarddrucker abrufen:

```
var server = new LocalPrintServer();
var queue = server.DefaultPrintQueue;
```

Seite drehen:

```
queue.UserPrintTicket.PageOrientation = PageOrientation.Landscape;
```

Eine reine Text-Seite erstellen:

```
TextBlock txt = new TextBlock();
txt.Text = "Hier ist eine Seite mit Fließtext. Hier ist eine Seite mit Fließtext.
```

BEISPIEL 16.9: Druckausgabe einer Textseite

```
Hier ist eine Seite mit Fließtext. Hier ist eine Seite mit Fließtext. Hier ist eine Seite mit
Fließtext. ";
```

Den *TextBlock* müssen Sie noch skalieren, sonst ist nichts zu sehen:

```
txt.Measure(new Size((double) queue.UserPrintTicket.PageMediaSize.Height, 200));
txt.Arrange(new Rect(0, 0, txt.DesiredSize.Width, txt.DesiredSize.Height));
```

Wer sich jetzt wundert, warum wir die Seitenhöhe statt der -breite angeben, sollte sich daran erinnern, dass Drehungen keinen Einfluss auf die *Height*- bzw. *Width*-Eigenschaft haben.

Hier erstellen wir den *XpsDocumentWriter*:

```
var writer = PrintQueue.CreateXpsDocumentWriter(queue);
```

Und damit landen unsere Ausgaben bereits auf dem Drucker:

```
writer.Write(txt);
```

Natürlich steht es Ihnen frei, auch mehrseitige Dokumente zu erstellen, die Sie an die *Write*-Methode übergeben (siehe dazu ab Seite 844). Beachten Sie allerdings, dass Sie nur einen Aufruf der *Write*-Methode realisieren können, Sie müssen als bereits ein komplettes mehrseitiges Dokument übergeben.

Damit wollen wir uns aus der Thematik "Druckausgabe" verabschieden und uns neuen Aufgabengebieten zuwenden, auch wenn sicher noch einiges zu diesem Thema zu sagen wäre.

Teil IV: Windows Store Apps

- **Erste Schritte**
- **WinRT-Oberflächen entwerfen**
- **Die wichtigsten Controls**
- **WinRT-Apps im Detail**
- **WinRT-Techniken**

Kapitel 17

Erste Schritte in WinRT

In diesem und den folgenden Kapiteln wollen wir Ihnen einen ersten Einblick in die neuesten Technologien aus dem Haus Microsoft geben, die für Sie als C#-Programmierer in den nächsten Jahren zunehmend interessant werden könnten.

Ausgehend von den Grundlagen der neuen Programmierschnittstelle *Windows Runtime* (kurz WinRT) zeigen wir Ihnen, wie Sie neben Ihren konventionellen Windowsanwendungen, basierend auf Windows Forms bzw. WPF, auch so genannte "Apps" für die neue Windows 8-Oberfläche[1] entwickeln können. Doch zunächst ein paar grundsätzliche Ausführungen, bevor wir uns mit den Feinheiten der WinRT-Entwicklung auseinandersetzen werden.

HINWEIS: Nach derzeitigem Stand wird WinRT nur ab Windows 8 verfügbar sein, eine Portierung auf Windows 7 etc. ist nicht angedacht.

17.1 Grundkonzepte und Begriffe

Statt vieler blumiger Worte (und Marketinggeblubber) hier eine kurze Übersicht der neuen Begriffe und Techniken, die Sie für die Entwicklung von Windows 8-Anwendungen benötigen.

17.1.1 Windows Runtime (WinRT)

Tja, was ist die Windows Runtime alias WinRT den nun eigentlich? Eine API, ein Framework oder eine Laufzeitumgebung? Die Antwort ist etwas vage und lautet "von allem etwas", denn es handelt sich um eine

- Kapselung des Betriebssystems und der Hardware (Funktion als Programmierschnittstelle),
- sowie um eine Laufzeitumgebung für die WinRT-Apps.

[1] Das Ganze hieß mal METRO, die neue Bezeichnung lautet Modern UI.

Die Aufgaben als Laufzeitumgebung sind recht vielfältig:

- Schnittstelle für das neue Windows 8-Userinterface,
- Installation der WinRT-Apps per Windows Store,
- Anzeige und Aktualisierung der Tiles[1] auf dem Windows 8-Screen,
- Überwachung des Lebenszyklus der Apps (WinRT-Anwendungen werden eigentlich nicht aktiv beendet, darum kümmert sich WinRT),
- Kapselung der Apps (Sandboxing),
- Datenaustausch zwischen den Apps per Contracts.

Intern ist WinRT eine echte Überraschung, wer hier zunächst auf ein erweitertes .NET-Framework tippt, liegt vollkommen falsch. Gehen Sie gedanklich 10 Jahre zurück und voilà: Sie sind wieder bei der guten alten COM-Technologie angekommen und programmiert ist das Ganze in C++!

Doch keine Sorge, auch hier ist die Zeit nicht stehengeblieben, dank neuer Konzepte (*Language Projection*, siehe Seite 868) können Sie die bekannten Unannehmlichkeiten bei der Arbeit mit COM-Objekten vergessen. Die gefürchtete DLL-Hölle (Shared Libraries) wird es nicht mehr geben, die zentralen WinRT-Klassen können nicht durch andere Anbieter "erweitert" werden, jede App bringt ihre eigenen Libraries mit und kann auch nur diese nutzen.

Weitere Details zum Thema finden Sie im Abschnitt 17.4 ab Seite 887.

17.1.2 Windows Store Apps

Nun ist der Begriff "App" schon mehrfach gefallen, und Sie haben sicher auch schon den Sinn erfasst. Dennoch wollen wir den Begriff noch einmal genauer "aufschlüsseln".

Mit "Windows Store **App**s" werden Anwendungen für Windows 8 bezeichnet, die aus der neuen Windows 8-Oberfläche heraus gestartet werden. Die folgende Abbildung zeigt Apps auf dem WinRT-Desktop in verschiedenen Tiles-Größen:

[1] Die sichtbaren Kacheln auf dem neuen Windows 8-Desktop

Diese Apps sind von der Darstellung und Verwendung auf die Bedienung per Touchscreen optimiert, können aber auch mit den konventionellen Eingabegeräten (Maus, Tastatur, Stift) bedient werden.

Basis dieser Apps ist WinRT und (im Falle von C# oder VB.NET als Entwicklersprache) eine Teilmenge des .NET-Frameworks. WinRT-Apps laufen sowohl auf der Intel- als auch auf der ARM-Plattform ab Windows 8.

HINWEIS: Diese Apps können, im Gegensatz zu den bekannten Desktop-Anwendungen, nur über den Windows-Store vertrieben bzw. installiert werden. Dazu wird ein App-Package erstellt, bei Microsoft eingereicht und geprüft und dann erst im Windows Store freigegeben. Selbstverständlich ist diese Infrastruktur nicht kostenlos.

Für Sie als C#-Programmierer interessant: Sie können Ihre XAML- und C#-Kenntnisse weiterverwenden[1], eine 1:1-Umsetzung von normalen WPF-Anwendungen ist jedoch meist nicht möglich (fehlende bzw. geänderte Namensräume, andere Controls, eingeschränkte Rechte).

17.1.3 Fast and Fluid

Hinter diesem so oft genannten Marketingbegriff versteckt sich die Erkenntnis, dass gerade im Bereich der Touch-Oberflächen vom Anwender eine sofortige Reaktion auf die Fingereingaben erwartet wird. Nichts ist wohl lästiger, als eine ruckelnde Oberfläche, die von vielen Anwendern intuitiv mangelnder Hardware-Leistung oder dem Betriebssystem angelastet wird.

[1] Alternativ besteht auch die Möglichkeit, diese Apps mit JavaScript und HTML5 zu erstellen.

Ursache ist jedoch meist der synchrone Ablauf unserer Programme, die in einigen Szenarien auch mal etwas länger brauchen. Beispielhaft seien hier nur Internet- und Dateizugriffe sowie umfangreiche Berechnungen genannt.

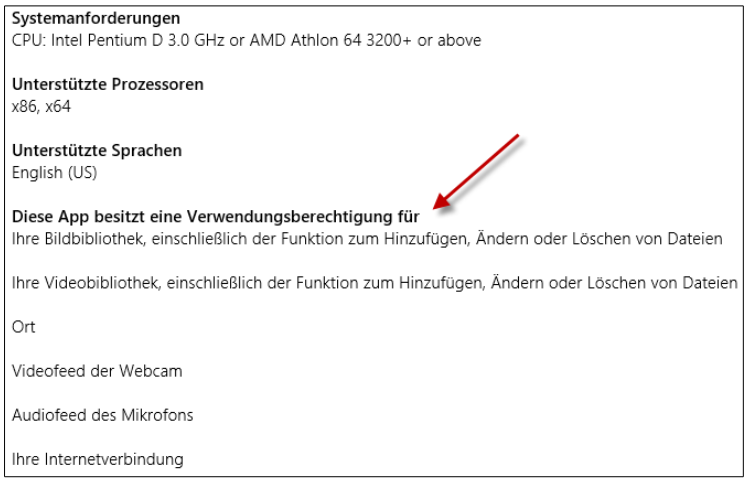
Mit Einführung der WinRT-Anwendungen hat sich Microsoft dieser Problematik angenommen. Statt es dem Programmierer zu überlassen, weite Teile des Programms mit einem mehr oder weniger unübersichtlichen Netz von Thread-Aufrufen zu überziehen, wurde das "Übel gleich an der Wurzel gepackt". Alle WinRT-Aufrufe, die länger als 50 Millisekunden dauern, sind asynchron programmiert, sodass die Programmoberfläche lebendig bleibt und der Nutzer das Gefühl einer sofortigen Reaktion hat. In diesem Zusammenhang sei auf Kapitel 9 verwiesen, wo wir uns bereits grundlegend mit den beiden Anweisungen *async* und *await* vertraut gemacht haben.

HINWEIS: Wer jetzt als Programmierer "der ersten Stunde" gequält an das mühsame Übergeben der Daten an den Vordergrundthread (Userinterface) denkt – vergessen Sie es! Alle Aktionen im Zusammenhang mit den beiden obigen Anweisungen laufen im Vordergrund-Thread ab, Sie können also direkt auf die Controls der Oberfläche zugreifen.

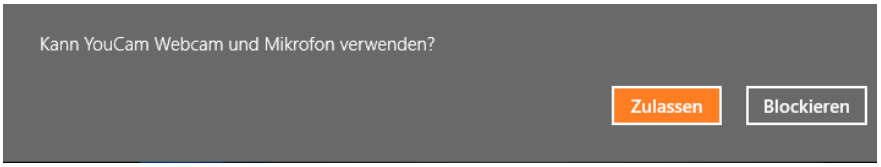
17.1.4 Process Sandboxing und Contracts

Mit der Umstellung auf eine WinRT-App werden Sie sich als Programmierer von vielen lieb gewordenen Freiheiten verabschieden müssen. Wer bereits Silverlight-Anwendungen programmiert hat weiß wovon die Rede ist: Ein schneller Zugriff auf die Festplatte oder die Registry, der Aufruf anderer Anwendungen, der Zugriff auf externe Geräte wie Webcam/Drucker oder das Nachladen von Daten aus dem Internet, all das ist prinzipiell nicht oder nur nach Zuteilung entsprechender Rechte durch den Anwender möglich.

Sie laden beispielsweise über den Windows Store die App "YouCam". Bereits im Store wird Ihnen angezeigt, welche Berechtigungen die App benötigt und für welche Plattformen diese verfügbar ist:



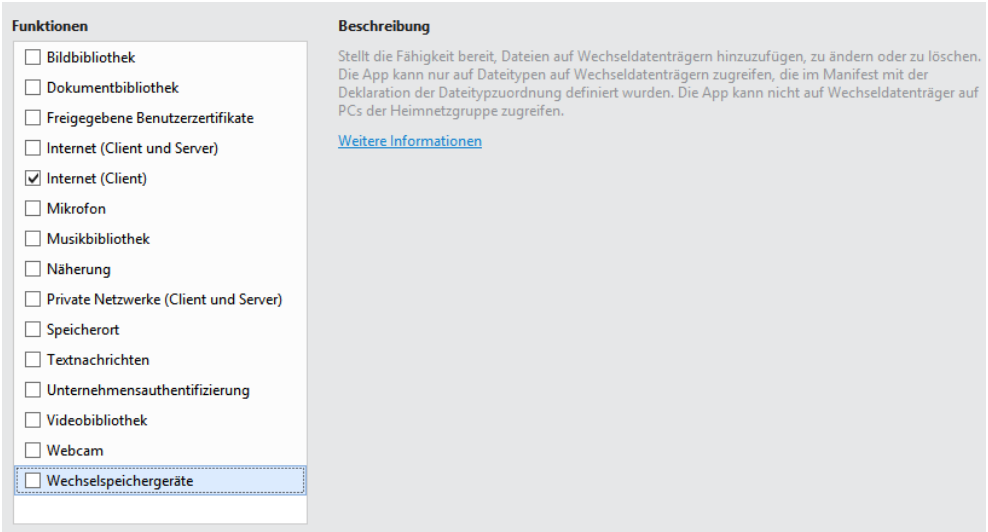
Genau an dieser Stelle kommen die so genannten *Contracts* ins Spiel. Jede WinRT-App definiert, welche Schnittstellen nach außen benötigt werden. Im obigen Fall wird unter anderem der Zugriff auf die Webcam und das Mikrofon benötigt. Beim ersten Start der App werden Sie als Anwender von WinRT (nicht von der Anwendung) gefragt, ob dies auch zulässig ist:



Nur bei zustimmender Antwort ist die Anwendung in der Lage, auf die Hardware bzw. die entsprechenden WinRT-Schnittstellen zuzugreifen. Realisiert wird dies über einen Proxy-Mechanismus, der sich quasi zwischen die App und die eigentlichen WinRT-Funktionen schiebt. Werden die gleichen WinRT-Funktionen hingegen aus einer normalen Desktop-Anwendung heraus aufgerufen, ist keine Freigabe erforderlich.

HINWEIS: Wer jetzt denkt, einfach entsprechende Funktionalitäten nachzuladen oder direkt auf den GAC (*Global Assembly Cache*) zuzugreifen – vergessen Sie es. Ihre Anwendung wird vor der Veröffentlichung im Windows Store geprüft und in diesem Fall nicht freigegeben. WinRT-Apps können nur auf die mitgelieferten Assemblies (aus dem Installations Package) und die WinRT-Assemblies zugreifen.

Doch wie legt man diese Einstellungen eigentlich fest? Hier hilft Ihnen die Anwendungsmanifestdatei weiter. Diese können Sie in Visual Studio komfortabel editieren (*Projekt|Store|Anwendungmanifest bearbeiten*):



HINWEIS: Die Einträge korrespondieren direkt mit den angezeigten Rechten im Windows Store.

Mehr zu den einzelnen Punkten finden Sie im Kapitel 20, welche eingeschränkten Möglichkeiten zum Dateizugriff bestehen zeigt Ihnen Abschnitt 20.3 ab Seite 1018.

17.1.5 .NET WinRT-Profil

Für die Ausführung Ihrer C#-WinRT-Apps ist neben der Windows Runtime (WinRT) zusätzlich eine Teilmenge des .NET Framework 4.5 erforderlich. Dieses steht auch auf der ARM-Plattform zur Verfügung.

Neben den aus Sicht des Programmierers problemlos verzichtbaren Teilen wie Console, WPF, ASP.NET sowie vielen doppelten APIs und Windows API-Wrappern, fehlen in diesem Profil jedoch auch Sachen, die Sie vermutlich vermissen werden:

- *System.Data* (Alternative: OData)
- *System.Web*
- LINQ to SQL etc.
- Remoting
- *System.Thread*
- AppDomains (jede WinRT-App hat nur eine AppDomain)

HINWEIS: Wollen Sie auf diese Schnittstellen/Technologien nicht verzichten, können Sie immer noch eine WPF-Anwendung schreiben, die zwar unter Windows 8 nicht als App unter WinRT, dafür aber auf dem Desktop läuft. Die ARM-Plattform bleibt Ihnen dann allerdings verschlossen.

17.1.6 Language Projection

Mit *Language Projection* bezeichnet Microsoft eine Technik, die es ermöglicht, die eigentlichen COM-basierten WinRT-Objekte auf das Typsystem der jeweiligen Programmiersprache abzubilden, d h. zu projizieren. Dies ist auch der Grund, dass bei WinRT auf die Verwendung von InterOp-Assemblies verzichtet werden kann. Nach dem Einbinden des Namenspace arbeiten Sie mit den WinRT-COM-Objekten wie mit normalen .NET-Klassen[1].

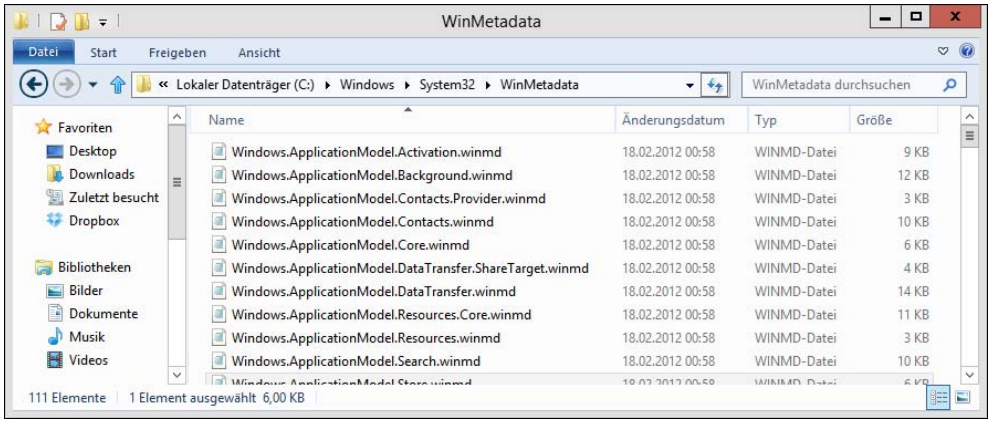
Die Language Projection funktioniert jedoch nicht nur in die bereits genannte Richtung, sondern auch wenn Sie als C#-Entwickler eigene WinRT-Klassen entwickeln, die z.B. von einem JavaScript- oder einem C++-Programm genutzt werden. Basis dieser Technologie sind umfangreiche Metadaten, die zu jeder Klasse, jedem Interface bzw. Member veröffentlicht werden. Der Compiler kümmert sich mit Hilfe dieser Informationen um die Umwandlung der Interfaces in Objekte, mappt

[1] In Einzelfällen stehen Erweiterungsmethoden zur Verfügung, z B. *AsStream* für die Umwandlung von *IInputStream* in *Stream*.

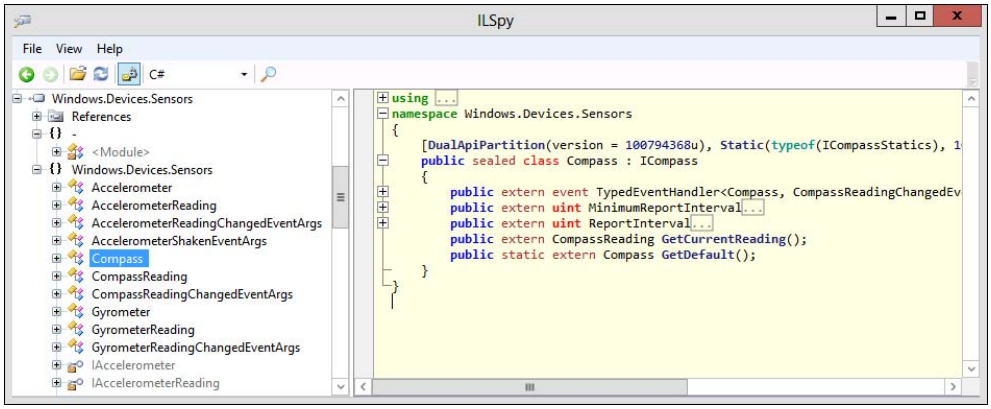
z.B. *HSTRING* auf *System.String,* wandelt die intern verwendeten *HRESULT*-Werte in C#-konforme Exceptions um (im Gegensatz zur Arbeit mit Win32-API-Funktionen können Sie wie gewohnt *try-except* einsetzen), mappt Methodennamen sprachspezifisch (z.B. *Add*, *Append* oder *append*) etc.

Als Entwickler müssen Sie sich also nicht an neue Bezeichner, Abläufe etc. gewöhnen, das nimmt ihnen alles der Compiler basierend auf den Metadaten der WinRT-Objekte ab.

Die Metadaten (*.*WinMD*-Dateien) finden Sie im Verzeichnis *C:\Windows\System32\WinMetadata*:



Ein Blick in diese Dateien ist zum Beispiel mit dem Tool *ILSpy*[1] möglich:



HINWEIS: Quellcode werden Sie hier nicht finden, es handelt sich eben nur um Metadaten und Interfaces.

[1] *http //sourceforge.net/projects/sharpdevelop/files/ILSpy/2.0/*

17.1.7 Vollbildmodus

Hier müssen wir Sie schonend darauf vorbereiten, dass Ihre WinRT-App zunächst grundsätzlich im Vollbildmodus auf dem primären Ausgabegerät läuft (WinRT-Apps unterstützen nur einen Screen). Frei bewegliche Fenster gibt es gar nicht, es ist immer nur ein Fenster je Anwendung aktiv und sichtbar.

Schlimmer noch – auch auf die gern und häufig eingesetzte *MessageBox* werden Sie prinzipiell verzichten müssen, Sie erinnern sich: die Oberfläche sollte "fast and fluid" sein, ein modales Window ist da nur im Weg[1].

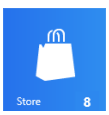
Vollbild bedeutet aber nicht, dass Sie den Anwender jetzt mit einer Fülle von Informationen "zupflastern" sollen. Ganz im Gegenteil: Weniger ist mehr, und so sollten nur relevante Daten im Fenster erscheinen. Für weitergehende Konfigurationen etc. stehen Ihnen konzeptionell einblendbare Navigationsleisten (oberer Bildschirmrand) oder Befehlsleisten (unterer Bildschirmrand) zur Verfügung[2].

Sollte die Bildschirmauflösung des Endgerätes ausreichend sein (>1366×768 Pixel), kann der Nutzer Ihre WinRT-App auch andocken, d h., diese nimmt die volle Höhe, aber nur zirka 1/3 der Breite des Bildschirms ein. Daneben wird dann maximal eine weitere WinRT-App dargestellt. Diesen Modus sollte Ihre Anwendung ebenfalls unterstützten, mit einem einfachen Zusammenschieben der Controls ist es hier aber nicht getan. Gleiches gilt für die Möglichkeit, unter Windows 8 (im Zusammenhang mit einem Tablet-PC) den Bildschirm im laufenden Betrieb zu drehen. Aus verfügbaren 1920x1200 Pixeln werden dann schnell mal 1200 x 1920 Pixel, Ihre App sollte auch darauf vorbereitet sein.

17.1.8 Windows Store

Wer hätte das gedacht, nachdem eine bekannte Firma (die mit dem Apfel) Wegbereiter dieses Trends war, will Microsoft natürlich auch nicht zurückstehen. Ab sofort (Windows 8) können Sie – nein besser: müssen Sie – Ihre Apps über den Windows Store beziehen. Dass Microsoft in diesem Fall einen nicht unbeträchtlichen Betrag (30%) für sich behält, dürfte klar sein. Dafür erhalten Sie dann geprüfte, sichere, bunte und tolle… (bla bla) Apps.

Aufrufen können Sie den Store über die entsprechende Kachel auf dem WinRT-Desktop:



Sollten Sie nicht bereits mit einer Windows Live ID in Windows eingeloggt sein, müssen Sie diese jetzt eingeben oder sich eine solche ID besorgen:

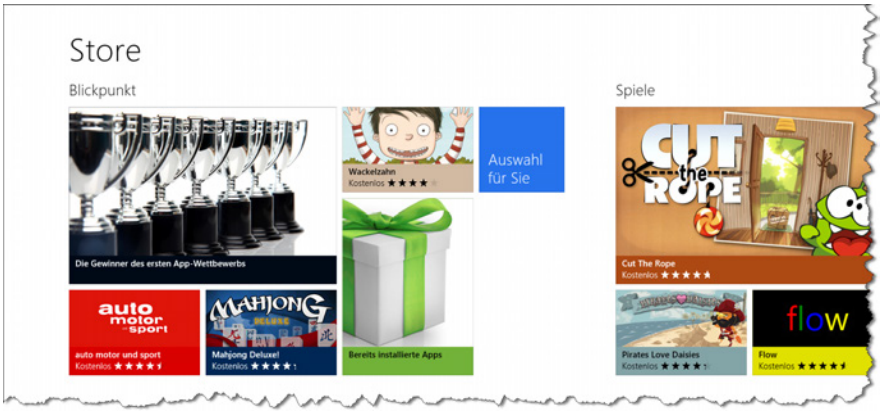
LINK: `https://login.live.com/`

---

[1] Kleiner Tipp: mit *MessageDialog* gibt es eine Alternative!

[2] Was Sie allerdings mit dem ganzen Platz (z.B. eines 27" oder 30" Monitors) anfangen ist dann Ihre Sache.

Nachfolgend steht Ihnen das komplette Angebot des Stores zur Verfügung. Neben Bewertungen und Verwendungsdetails wird Ihnen eine Vorschau sowie eine Beschreibung der App angezeigt. Mit dem Klick auf die Schaltfläche *Installieren* wird der Installationsvorgang angestoßen, die Fertigstellung wird durch eine kleine Textnachricht auf dem WinRT-Desktop gemeldet. Über die entsprechende Kachel auf dem WinRT-Desktop lässt sich die App nachfolgend auch wieder deinstallieren (rechte Maustaste, untere Befehlsleiste).



Für Sie als Entwickler interessant:

- Sie müssen eine Entwicklerkonto eröffnen (ab 50$ für eine Einzelperson).
- Microsoft prüft die Anwendung auf Einhaltung der Richtlinien[1] und behält sich eine Löschung der App vor (das betrifft im Zweifelsfall auch die Endgeräte beim Endanwender).
- Apps werden im Allgemeinen als Lizenz zum Betrieb auf **fünf** Endgeräten angeboten, Sie verkaufen also keine Einzellizenzen.

Weitere Details finden Sie unter der folgende Adresse:

LINK: `http://windows microsoft.com/de-DE/windows/store-terms-of-use`

17.1.9 Zielplattformen

Hier findet sich eine der wesentlichsten Neuerungen: WinRT wird nicht nur für die x86- bzw. x64-Prozessorarchitektur verfügbar sein, sondern auch für ARM-Prozessoren. Dies dürfte auch der wohl wesentlichste Grund für die Entwicklung von WinRT gewesen sein, das .NET-Framework auf ARM zu portieren wäre wohl zu aufwändig gewesen.

WinRT ist auch für Nicht-WinRT-Apps nutzbar, d.h. in Ihren konventionellen Anwendungen. Dies ist aber eher ein fragwürdiges Feature, da Sie sich damit voraussichtlich ausschließlich auf Windows 8 als Zielplattform beschränken.

[1] Diese sind recht detailliert und beschreiben beispielsweise auch, welchen maximalen Datentransfer eine App im getakteten Modus (kostenpflichtige Mobilverbindung) haben darf, welche Startzeit maximal erlaubt ist ...

17.2 Entwurfsumgebung

Haben Sie sich für die Entwicklung von WinRT-Apps entschieden, sollten Sie auch einen Blick auf die verfügbaren Tools bzw. auf die Grundvoraussetzungen der WinRT App-Entwicklung werfen.

17.2.1 Betriebssystem

Gleich ein wichtiger Hinweis vorweg:

HINWEIS: WinRT-Apps können Sie nur unter Windows 8 entwickeln, d h., Sie brauchen in jedem Fall einen Windows 8-PC.

Von der Verwendung einer virtuellen Maschine (VMWare/VirtualBox) raten wir ab, zum einen stimmt die Performance nicht, zum anderen ist die Bedienung dank neuer Mausgesten etc. eine Qual[1]. Wer noch vorsichtig ist und noch nicht komplett auf Windows 8 umsteigen will, kann es mit einer virtuellen Festplatte auf einem Windows 7-System versuchen.

HINWEIS: Wir, die Autoren, übernehmen keine Gewähr und Haftung für die folgende Anleitung. Sichern Sie zunächst Ihre Daten, um einem Datenverlust vorzubeugen.

1. Erzeugen Sie unter Windows 7 per Datenträgerverwaltung eine neue virtuelle Festplatte (zirka 100 GB dynamisch). Legen Sie diese beispielsweise unter dem Namen *c:\Win8Disk.vhd* ab.
2. Legen Sie die Windows 8-DVD ein und starten Sie den Installationsprozess (nicht aus Windows heraus, sondern beim Systemstart). Wählen Sie im Laufe der Installation zunächst noch **kein** Ziellaufwerk aus, sondern rufen Sie an dieser Stelle mit *Shift-F10* das Konsolenfenster auf.
3. Starten Sie an der Konsole die Anwendung *Diskpart* und geben Sie die folgenden Anweisungen ein:

```
DISKPART> select vdisk file=c:\Win8Disk.vhd
DISKPART> attach vdisk
```

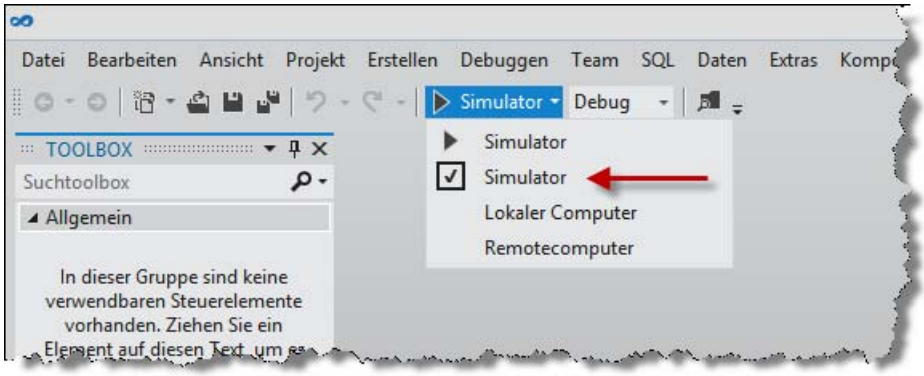
4. Wechseln Sie per *Alt+Tab* in das Fenster zur Auswahl des Ziellaufwerks und aktualisieren Sie die Liste. Die neue Partition sollte jetzt in der Liste angezeigt werden.
5. Wählen Sie die neue Partition aus und ignorieren Sie die Warnung, dass auf dieser Partition kein System installiert werden kann.
6. Setzen Sie die Installation fort. Nach dem Systemneustart erwartet Sie eine Auswahl zwischen Windows 8 und Ihrem alten Windows 7-System.
7. Unter Windows 8 können Sie jetzt Visual Studio installieren und WinRT-Apps entwickeln, ohne dass Sie Ihr altes System verändern müssen.

[1] Treffen Sie in der virtuellen Maschine mal genau die Ecken bzw. den oberen Bildschirmrand.

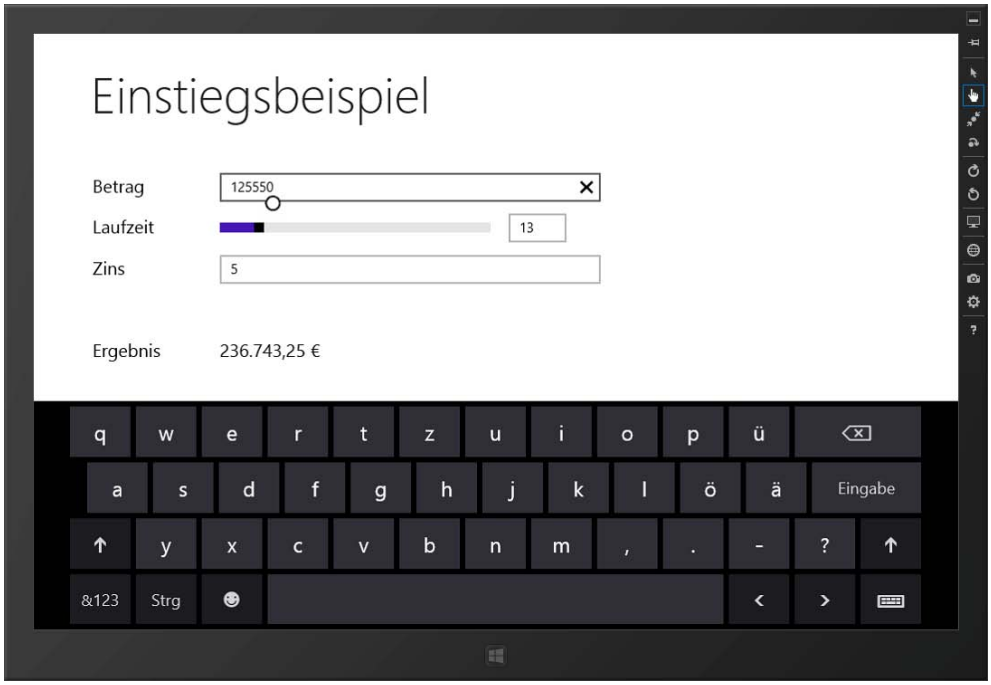
17.2.2 Windows-Simulator

Tablet-Entwicklung hin oder her, aber wollen Sie gleich ein oder mehrere Geräte für 500-1000 Euro kaufen, nur um ein paar Mal eine WinRT-App unter den Bedingungen eines Tablets zu testen? Sicher nicht, es sei denn, Sie wissen schon vorher, dass Ihre App der absolute Renner wird.

Hier hilft Ihnen der Windows-Simulator weiter. Aktivieren Sie diesen über die Symbolleiste:

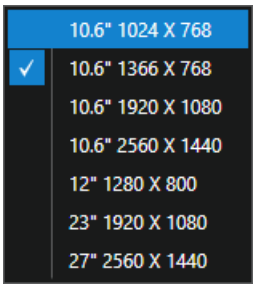


Starten Sie jetzt ein Projekt, wird dieses nicht auf dem normalen Windows 8-Desktop, sondern im Simulator wiedergegeben:

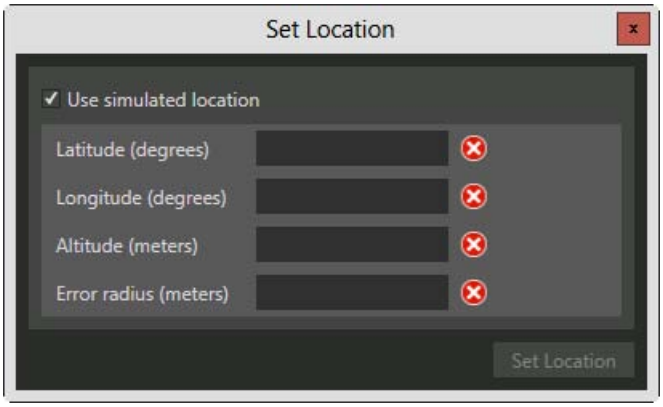


Dies hat gleich mehrere Vorteile:

- Sie können die Reaktion auf diverse Fingereingaben (Drehen, Zoom etc.) testen. Dazu wird mit der Maus eine Fingereingabe simuliert.
- Sie können die Verwendung einer Bildschirmtastatur testen.
- Sie können das Verhalten der Anwendung bei unterschiedlichen Geräteauflösungen (1024x768 bis 2560x1440 Pixel) simulieren:



- Der Bildschirm (bzw. das Gerät) kann gedreht werden. Ihr Programm sollte sinnvoll auf diese Änderung reagieren.
- Sie können dem Gerät einen Standort zuweisen (nützlich für Anwendungen, die diese Daten auswerten):



- Sie können Screenshots des aktuellen Inhalts anfertigen und diese in die Zwischenablage oder als Datei speichern lassen.

HINWEIS: Last but not least, sollten Sie auf einen finalen Test Ihrer App unter realen Bedingungen nicht verzichten. Insbesondere die Bedienung per Hand kann wohl nur im echten Praxistest realistisch geprüft werden[1].

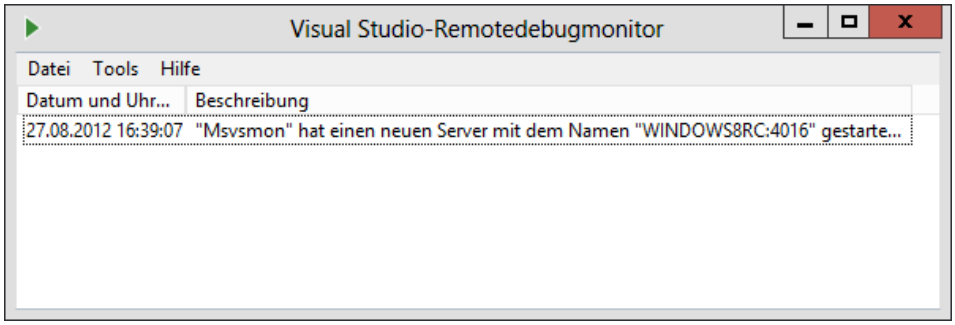
[1] Nicht jeder hat so lange Finger, dass diese gleichzeitig zum Halten und Tippen auf dem Bildschirm taugen.

17.2.3 Remote-Debugging

Ein zweites hilfreiches Werkzeug für den WinRT-Entwickler ist der Remotedebugger. Diesen können Sie unter folgender Adresse herunterladen:

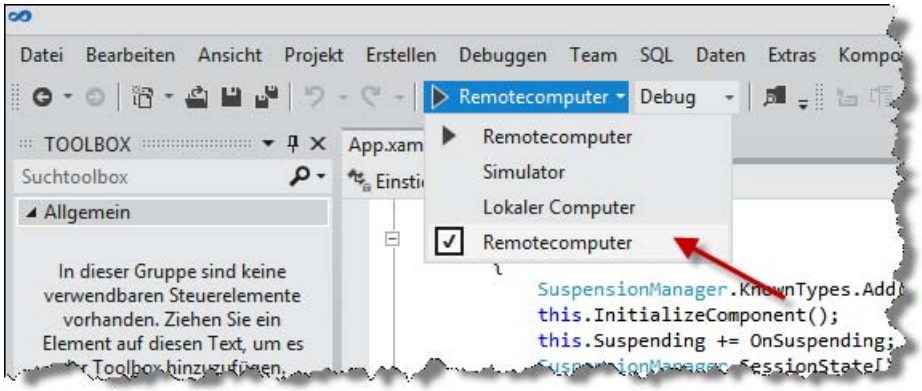
LINK: `http://www microsoft.com/visualstudio/deu/downloads`

Installieren Sie diesen auf einem weiteren Windows 8-System[1] (im Idealfall gleich ein Intel-Tablet) und starten Sie den Remote-Debugger:



HINWEIS: Als recht praktisch hat sich in diesem Zusammenhang die Verwendung eines Convertible Tablet PCs erwiesen. Sie können auf diesem System Ihren Fingern freien Lauf lassen und den Bildschirm beschmieren, gleichzeitig ist aber auch die Arbeit mit Maus und Tastatur möglich.

In Visual Studio müssen Sie jetzt noch den Remotecomputer als Ausgabeziel festlegen:



Falls dieser nicht gefunden wird, geben Sie dessen Adresse einfach per Hand ein.

[1] Auch hier sind Sie nicht davon befreit, ein Windows 8-System zu nutzen, andernfalls steht das WinRT-Framework ja nicht zur Verfügung.

Der Vorteil dieser Debugging-Variante: Sie können an Ihrem leistungsfähigen System arbeiten und entwickeln und trotzdem die Anwendung auf einem wesentlich leistungsschwächeren System testen und debuggen. Dabei gehen Sie genau wie bei der lokalen Variante vor, setzen Ihre Breakpoints etc. und können dann Schritt für Schritt den Code prüfen.

HINWEIS: Von einer WinRT-Besonderheit werden Sie allerdings auch beim Remote-Debugger nicht verschont: Für das Starten von Apps und den Eintritt der Anwendung in den Suspend-Modus gelten unter WinRT strikte Zeitvorgaben. Werden diese, z.B. durch einen gesetzten Breakpoint, nicht eingehalten, beendet WinRT Ihre App. In diesen zeitkritischen Fällen sollten Sie Zwischenergebnisse am besten per *Debug*-Objekt im Ausgabefenster anzeigen.

17.3 Ein (kleines) Einstiegsbeispiel

Nun wurde schon seitenweise über WinRT herumgelabert und Sie finden immer noch kein praktisches Beispiel! Dem wollen wir jetzt schnell abhelfen.

HINWEIS: Zu den Grundlagen der Oberflächengestaltung per XAML und der Datenbindung (siehe Kapitel 12, 13 bzw. 15) wollen wir uns hier nicht komplett wiederholen.

17.3.1 Aufgabenstellung

Wir wollen eine einfache App zur Zins-Berechnung entwickeln. Dazu benötigen wir lediglich drei Eingabefelder, d h. für

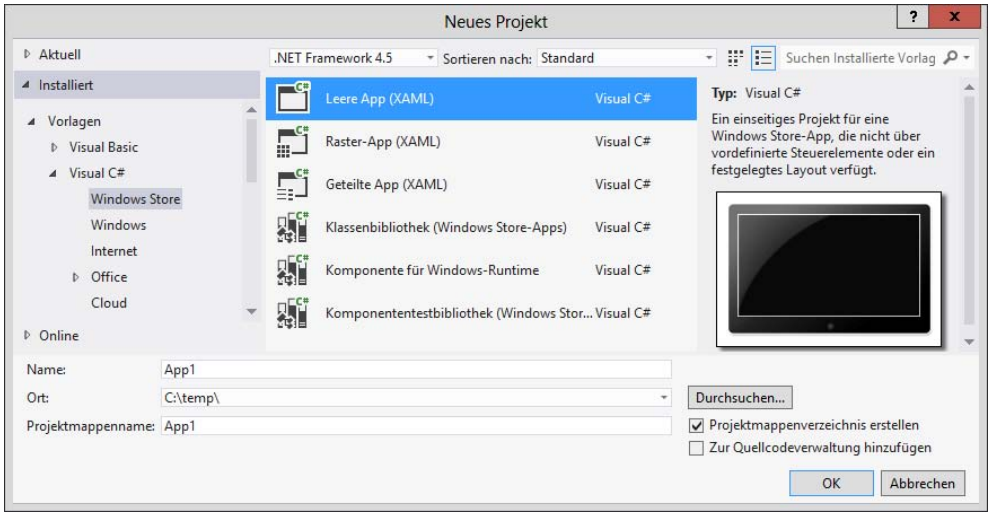
- den Anlagebetrag,
- den Zins
- und die Laufzeit.

Einfachheitshalber gehen wir von einer jährlichen Zahlweise aus. Ergebnis unserer Bemühungen ist der Endbetrag nach obiger Laufzeit.

17.3.2 Quellcode

Im Gegensatz zu vielen anderen Beispielprogrammen wollen wir uns zunächst mit der eigentlichen Logik beschäftigen, bevor wir die dazugehörige Oberfläche entwerfen. Der Vorteil: wir können die gleiche Logik auch für eine WPF-Desktop-Anwendung nutzen, es sind keinerlei Änderungen erforderlich.

Starten Sie Visual Studio, klicken Sie auf *Datei|Neu|Projekt* und wählen Sie als neuen Projekttyp in der Rubrik *Windows Store* die Option *Leere App*:



Vergeben Sie den Namen *Einstieg*.

Wer jetzt glaubt, wie bei den Windows Forms ein fast leeres Projekt vorzufinden, sieht sich getäuscht. Neben diversen Projektdateien ist auch schon ein kleines Framework vorhanden, das sich um die wichtigsten Aufgaben im Lebenszyklus der WinRT-App kümmert:

- "Geburt": Start durch Klick auf die Kachel oder Aufruf durch einen Contract
- "Schlaf": Suspend-Mode, wenn die Anwendung im Hintergrund liegt
- Der "Tod" bzw. das aktive Ende[1] spielt im "Leben" der WinRT-App aus Programmierersicht keine Rolle, da alle in diesem Zusammenhang relevanten Aufgaben schon beim Suspend erledigt sein müssen.

HINWEIS: Um die vielen Projektdateien und deren Bedeutung kümmern wir uns an dieser Stelle zunächst nicht.

Nächster Schritt ist das Erstellen einer Klasse *Finanz*, die uns neben den vier Eigenschaften *Betrag*, *Zins*, *Laufzeit* und *Endwert* noch eine interne Berechnungsmethode *Calc* zur Verfügung stellt. Soll die Klasse (bzw. deren Instanz) an die Oberfläche gebunden werden, ist, wie auch bei WPF, die Schnittstelle *INotifyPropertyChanged* zu unterstützen. Die Aktualisierung der gebundenen Oberflächenelemente nach einer Wertänderung ist Aufgabe des *PropertyChanged*-Eventhandlers:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

[1] ... durch WinRT (Speichermangel) oder den Anwender (Wischbewegung von oben)

```
namespace Einstieg
{
    public class Finanz : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
```

Die vier Eigenschaften:

```
        double betrag = 0;
        public double Betrag
        {
            get { return betrag; }
            set
            {
                betrag = value;
                Calc();
            }
        }
        double zins = 0;
        public double Zins
        {
            get { return zins; }
            set
            {
                zins = value;
                Calc();
            }
        }

        int laufzeit = 0;
        public int Laufzeit
        {
            get { return laufzeit; }
            set
            {
                laufzeit = value;
                Calc();
            }
        }
```

Der Endwert ist aus naheliegenden Gründen schreibgeschützt:

```
        double endwert = 0;
        public double Endwert
        {
           get { return endwert; }
        }
```

Eine zweite Variante dieser Eigenschaft liefert uns gleich einen formatierten *String* (Währungswert) statt des *Double*-Wertes:

```
        public string EndwertAsString
        {
            get
            {  return endwert.ToString("c"); }
        }
```

Die Berechnung des Endwertes:

```
        private void Calc()
        {
            try
            {
                endwert = betrag * Math.Pow(1 + zins / 100, laufzeit);
            }
            catch (Exception)
            {
                endwert = 0;
            }
```

Hier lösen wir auch das *PropertyChanged*-Ereignis aus:

```
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs("Endwert"));
                PropertyChanged(this, new PropertyChangedEventArgs("EndwertAsString"));
            }
        }
    }
}
```

Jede Änderung einer Eigenschaft hat eine neue Berechnung und damit die Aktualisierung des Endwertes zur Folge.

Damit sind alle Vorbereitungen abgeschlossen und wir können uns der Oberfläche zuwenden.

17.3.3 Oberflächenentwurf

Öffnen Sie die Datei *MainPage.xaml*, dies ist das "Startformular" unserer App. Dabei handelt es sich um eine Instanz der Klasse *Page* (siehe Code-behind-Datei *MainPage.xaml.cs*).

HINWEIS: Die erste sichtbare Page legen Sie in der Datei *App.xaml.cs* in der Methode *OnLaunched* fest. Dies ist auch der einzig mögliche Eintrittspunkt für Ihr Programm.

Doch zurück zu *MainPage.xaml*. Erweitern Sie bitte den XAML-Code um die im Folgenden fett hervorgehobenen Teile:

```
<Page
    x:Class="Einstieg.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:local="using:Einstieg"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
```

Unsere Seitenüberschrift (einen Formularkopf gibt es nicht) positionieren wir per *Margin*, die Schriftformatierung übernimmt in diesem Fall die statische Ressource *PageHeaderTextStyle* aus der Datei *StandardStyles.xaml*:

```
<TextBlock HorizontalAlignment="Left" Margin="65,48,0,0" TextWrapping="Wrap"
          Text="Anfängerbeispiel" VerticalAlignment="Top"
          Style="{StaticResource PageHeaderTextStyle}"/>
```

Mit Hilfe eines *Grid*-Controls positionieren wir die Eingabe- und Beschriftungsfelder:

```
<Grid HorizontalAlignment="Left" Height="227" Margin="65,147,0,0"
      VerticalAlignment="Top" Width="563">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="3*"/>
    </Grid.ColumnDefinitions>
```

Die einzelnen Beschriftungen per *TextBlock*-Controls:

```
    <TextBlock Grid.Column="0" Grid.Row="0" HorizontalAlignment="Left" Text="Betrag"
               VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Column="0" Grid.Row="1" HorizontalAlignment="Left" Text="Laufzeit"
               VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Column="0" Grid.Row="2" HorizontalAlignment="Left" Text="Zins"
               VerticalAlignment="Center" FontSize="20"/>
    <TextBlock Grid.Column="0" Grid.Row="4" HorizontalAlignment="Left" Text="Ergebnis"
               VerticalAlignment="Center" FontSize="20"/>
```

Die Eingabefelder binden wir direkt an die Eigenschaften der *Finanz*-Klasse (die Instanz selbst binden wir erst zur Laufzeit per *DataContext*-Eigenschaft der Page):

```
    <TextBox Grid.Column="1"  Grid.Row="0" Text="{Binding Path=Betrag, Mode=TwoWay}"
               VerticalAlignment="Center" />
    <TextBox Grid.Column="1"  Grid.Row="1" Text="{Binding Path=Laufzeit, Mode=TwoWay}"
               VerticalAlignment="Center" />
    <TextBox Grid.Column="1"  Grid.Row="2" Text="{Binding Path=Zins, Mode=TwoWay}"
               VerticalAlignment="Center" />
```

HINWEIS: Sie müssen den Binding-Mode auf "TwoWay" festlegen, andernfalls wird das gebundene Objekt bei Textänderungen nicht aktualisiert.

Den berechneten Endwert geben wir direkt als Zeichenkette aus, so haben wir direkten Einfluss auf die Formatierung[1]:

```
            <TextBlock Grid.Column="1" Grid.Row="5" HorizontalAlignment="Left"
                       Text="{Binding Path=EndwertAsString}"  VerticalAlignment="Center"
                       FontSize="20" Name="tb1"/>
        </Grid>
    </Grid>
</Page>
```

Das war es bereits, bleibt nur noch die Aufgabe, eine Instanz der *Finanz*-Klasse zu erzeugen und dem *DataContext* der Seite zuzuweisen.

Öffnen Sie dazu zunächst die Datei *App.xaml.cs* und nehmen Sie folgende Erweiterung vor:

```
...
namespace Einstieg
{
    sealed partial class App : Application
    {
        static public Finanz finanzrechner = new Finanz();
...
```

HINWEIS: Den Grund für die Definition in der statischen *App*-Klasse erfahren Sie in den folgenden Abschnitten, deshalb bitte etwas Geduld.

Die eigentliche Datenbindung realisieren wir in der Datei *MainPage.xaml.cs*:

```
...
    public sealed partial class MainPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
            this.DataContext = App.finanzrechner;
        }
...
```

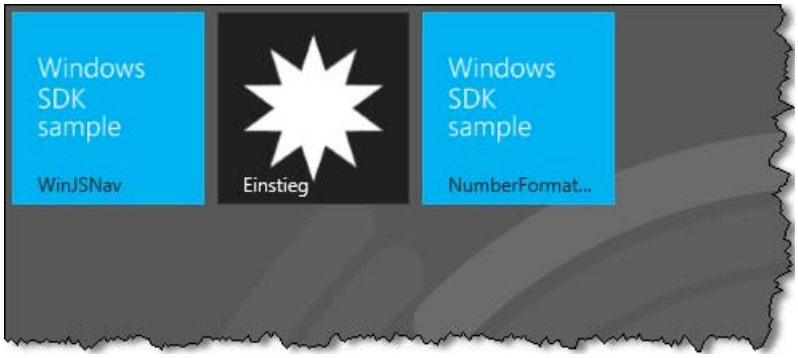
17.3.4 Installation und Test

Die Zeit ist reif für einen ersten Test. Klicken Sie auf F5. Nach kurzem Aufflackern des Splash-Screens sollte die voll funktionstüchtige Oberfläche auf dem Desktop erscheinen:

[1] In der vorliegenden Version war noch keine Stringformatierung wie in WPF implementiert.

So weit – so gut, das gleiche Programm hätten Sie mit minimalen Änderungen auch als WPF-Anwendung realisieren können.

Doch lassen Sie sich nicht täuschen, Visual Studio hat die Anwendung nicht nur kompiliert, sondern ganz nebenbei auch gleich mit einer Entwicklerlizenz für die WinRT-Oberfläche installiert (dies ist der Windows Runtime geschuldet):



17.3.5 Verbesserungen

Wie schon erwähnt, verhalten sich WinRT-Apps etwas anders als ihre WPF- oder Windows Forms-Kollegen. Ein wichtiger Unterschied ist der Wechsel der Vordergrundanwendung. In diesem Fall wird die bisher aktive WinRT-App nach einer Karenzzeit in den Suspend-Modus geschickt, in welchem bis auf den Speicher keine Ressourcen mehr belegt werden. Für Sie als Programmierer wichtig: bereits hier müssen Sie sich darum kümmern, Daten zu sichern und gegebenenfalls bei einem Neustart wieder herzustellen.

Persistenz

Was bedeutet dies für unsere Anwendung? Schaltet der Nutzer bei laufender Anwendung zu einer anderen App um und wird im weiteren Verlauf mehr Speicher benötigt, beendet WinRT Ihre App. Alle bisherigen Eingaben gehen verloren. Was tun? Hier können Sie sich mit einer "selbstgestrickten" Lösung behelfen, oder Sie werfen einen Blick in die vielen verfügbaren Microsoft-Beispiele.

In fast allen Projekten findet sich die statische Klasse *SuspensionManager*, deren Aufgabe das Speichern und Wiederherstellen des Anwendungsstatus ist[1].

Kopieren Sie diese Datei in Ihr Projekt und erweitern Sie die Datei *App.xaml.cs* wie folgt:

```
...
    sealed partial class App : Application
    {
        static public Finanz finanzrechner = new Finanz();

        public App()
        {
```

Zunächst müssen wir dem *SuspensionManager* unsere eigene Klasse "bekannt machen"[2]:

```
            SuspensionManager.KnownTypes.Add(typeof(Finanz));
            this.InitializeComponent();
            this.Suspending += OnSuspending;
```

Hier weisen wir den *SuspensionManager* an, unsere Instanz als Session-Variable zwischenzuspeichern (Voraussetzung für die spätere Persistenz):

```
            SuspensionManager.SessionState["io"] = App.finanzrechner;
```

Da es sich um eine Objekt handelt (wird als Verweis gespeichert) können wir die Zuweisung bereits hier realisieren, auch wenn sich später die Eigenschaften durch Nutzereingaben ändern.

```
        }
```

Auf den Suspend-Modus (bzw. die Vorbereitung dafür) können Sie mit der folgenden Methode reagieren, die bereits vordefiniert ist:

```
        async void OnSuspending(object sender, SuspendingEventArgs e)
        {
```

Das *SuspendingDeferral*-Objekt informiert das System, dass Ihre Anwendung sich um die Vorbereitung des Suspend-Modus kümmert, mit der Methode *Complete* zeigen Sie an, dass Ihre App für den Suspend-Modus bereit ist:

```
            SuspendingDeferral deferral = e.SuspendingOperation.GetDeferral();
```

Hier rufen wir die Methode *SaveAsync* des *SuspensionManagers* auf, diese speichert die Daten im App-Verzeichnis als XML-Datei:

```
            await SuspensionManager.SaveAsync();
```

[1] Wie diese Klasse intern realisiert ist, soll uns an dieser Stelle nicht interessieren, wir nutzen diese Klasse nur.

[2] Dies ist für die Serialisierung/Deserialisierung erforderlich.

```
            deferral.Complete();
        }
```

Wird die Anwendung wieder aktiviert, können Sie zunächst auswerten, aus welchen Gründen dies erfolgt ist:

```
        async protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
```

Wurde die Anwendung aus irgendwelchen Gründen beendet, stellen wir den vorherigen Zustand wieder her, indem wir die Persistenzdaten per *SuspensionManager* aus dem App-Verzeichnis laden und der Instanz der *Finanz*-Klasse zuweisen.

```
            if (args.PreviousExecutionState == ApplicationExecutionState.Terminated
             || args.PreviousExecutionState == ApplicationExecutionState.ClosedByUser)
            {
                await SuspensionManager.RestoreAsync();
                if (SuspensionManager.SessionState.ContainsKey("io"))
                    finanzrechner = SuspensionManager.SessionState["io"] as Finanz;
            }
            var rootFrame = new Frame();
            rootFrame.Navigate(typeof(MainPage));
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
    }
}
```

HINWEIS: Sie können natürlich weitere Objekte bzw. Eigenschaften mit Hilfe des SuspensionManagers speichern, beispielsweise das aktive Eingabefeld oder bei größeren Anwendungen die aktuelle Seite. Für den Endanwender soll das Gefühl aufkommen, die Anwendung nie neu gestartet, sondern nur neu aktiviert zu haben. Ein zwischenzeitliches Beenden soll möglichst unbemerkt bleiben.

Starten Sie die App erneut, geben Sie einige Werte ein und beenden Sie die Anwendung mit der Wisch-Geste von oben. Nach einem Neustart sollten alle Eingabewerte wieder angezeigt werden, so als ob die App nie beendet worden wäre.

Touchscreen

Haben Sie die App auf dem Arbeitsplatz-PC getestet, wird es Ihnen kaum aufgefallen sein, aber die Eingabe von Zahlenwerten auf einem Tablet-PC ist nicht unbedingt komfortabel. Aus diesem Grund bietet es sich beispielsweise an, die Laufzeit nicht per *TextBox*, sondern auch per *Slider* einzugeben. Ändern Sie dazu *MainPage.xaml* wie folgt:

```
...
            <TextBox Grid.Column="1"  Grid.Row="1" Text="{Binding Path=Laufzeit, Mode=TwoWay}"
                     VerticalAlignment="Center" />
...
```

zu

```
...
           <StackPanel Orientation="Horizontal" Grid.Column="1"  Grid.Row="1">
               <Slider Value="{Binding Path=Laufzeit, Mode=TwoWay}" VerticalAlignment="Center"
               Width="300" />
               <TextBox  Text="{Binding Path=Laufzeit, Mode=TwoWay}" VerticalAlignment="Center"
               Margin="20,0,0,0" />
           </StackPanel>
...
```

Nach einem erneuten Start werden Sie feststellen, dass sich zwar der Endwert ändert, egal, ob Sie die *TextBox* oder den *Slider* für die Eingabe nutzen, allerdings laufen die beiden Steuerelemente nicht synchron. Dazu müssen wir einen kleinen Eingriff in der *Finanz*-Klasse vornehmen:

```
public int Laufzeit
       {
           get
           {  return laufzeit; }
           set
           {
               laufzeit = value;
               Calc();
               if (PropertyChanged != null)
                   PropertyChanged(this, new PropertyChangedEventArgs("Laufzeit"));
           }
       }
```

Wir benachrichtigen die angeschlossenen Controls, wenn sich die Laufzeit ändert. Nach einem Neustart sollte die Anwendung wie gewünscht funktionieren, Änderungen am *Slider* sollten sich auf die *TextBox* auswirken und umgekehrt.

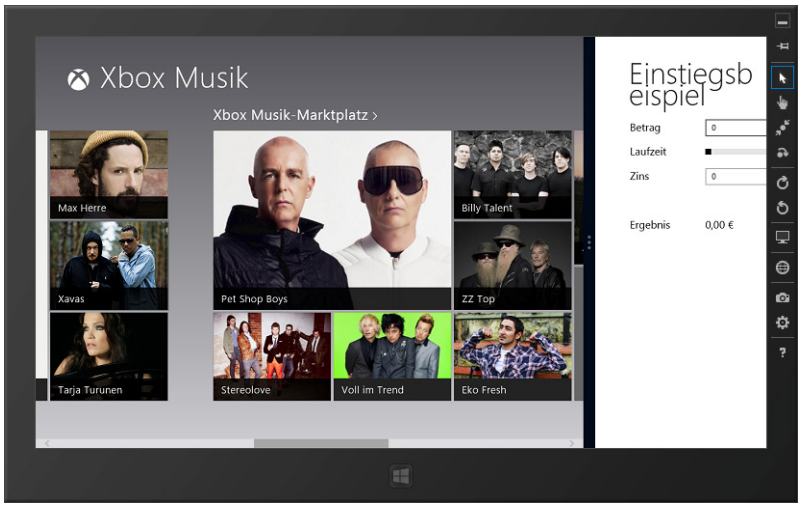
17.3.6 Fazit

Eigentlich sollte unser Beispielprogramm kurz sein, aber Sie haben sicher schon gemerkt, dass es unter WinRT mit einem einfachen Eingabeformular meist nicht getan ist. Neben der Persistenz müssen Sie ganz nebenbei auch ein Auge auf das Oberflächendesign werfen. Im Falle unserer Anwendung haben wir ja schon etwas vorausschauend ein *Grid* mit dynamischen Spaltenbreiten gewählt:

```
            <Grid.ColumnDefinitions>
               <ColumnDefinition Width="*"/>
               <ColumnDefinition Width="3*"/>
           </Grid.ColumnDefinitions>
```

Prüfstein für Ihre Anwendung sollte in jedem Fall der Simulator in der niedrigsten Auflösung im Hochkant-Format sein. In dieser Konstellation steht am wenigsten Platz in horizontaler Richtung zur Verfügung:

Doch nicht genug der Pein – unter WinRT kann eine App auch angedockt werden:



Spätestens hier macht unsere App (siehe rechter Rand) keine "gute Figur" mehr[1], wir müssten explizit auf diesen Modus reagieren und ein geändertes Layout anbieten. Doch dies soll nicht Gegenstand dieses Einführungsbeispiels sein.

[1] Dieses Schicksal teilt unsere App jedoch auch mit fast allen Microsoft-Beispielen, im angedockten Modus ist nun mal in den wenigsten Fällen noch genug Platz.

17.4 Weitere Details zu WinRT

Nachdem wir einen schmalen Pfad durch den Dschungel der Neuerungen geschlagen und uns mit einem ersten Beispiel herumgeplagt haben, wollen wir nun einen etwas detaillierteren Blick auf die neue Windows Runtime und deren Konzepte werfen.

Wir hatten bereits im ersten Abschnitt festgestellt, dass WinRT zum einen

- ein Interface bzw. eine Schnittstelle zu den Systemfunktionen und
- zum anderen eine Framework zur Installation und Verwaltung der WinRT-Apps ist.

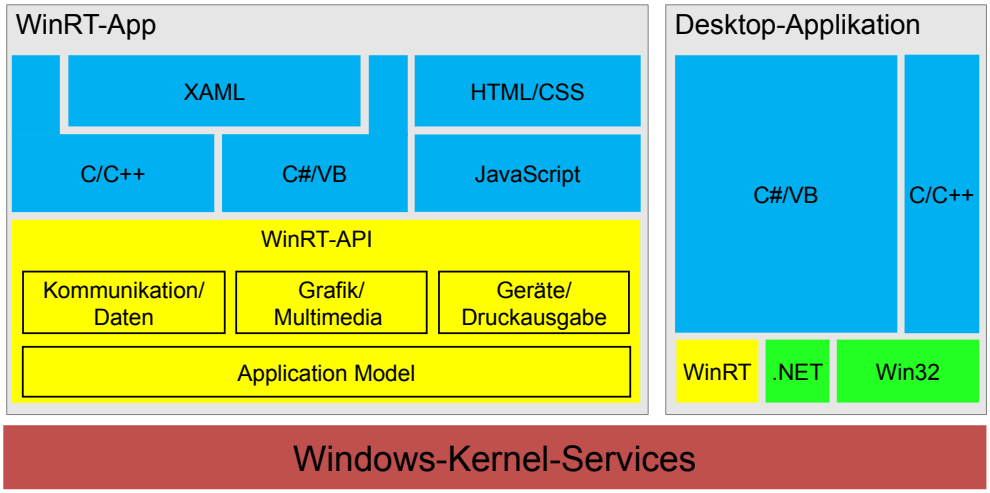
Doch welche Rolle spielt WinRT für Sie als C#-Programmierer?

17.4.1 Wo ist WinRT einzuordnen?

Als Programmierer werden Sie sicher wissen wollen, wie und an welcher Stelle Sie WinRT nutzen können bzw. müssen. Grundsätzlich bieten sich zwei Einsatzszenarien an:

- Windows Store-Apps (der Normalfall)
- Desktop-Apps

Die folgende Abbildung zeigt, welche Bedeutung WinRT für den jeweiligen Anwendungszweck hat:



Während WinRT bei WinRT-Apps die unbedingt Voraussetzung für deren Ausführung und Installation ist, stellt WinRT im Desktopbereich lediglich eine Alternative zu den bisherigen Schnittstellen .NET-Framework und Win32-API dar. Ein kleines Beispiel für die einfache Verwendung einer WinRT-Klasse in einer konventionellen WPF-Anwendung finden Sie im Praxisbeispiel

► 17.6.1 WinRT in Desktop-Applikationen nutzen

HINWEIS: Wie schon erwähnt, sollten Sie WinRT bei Desktop-Anwendungen nur in Ausnahmefällen verwenden (Zugriff auf Webcam, Sensoren, Medien etc.). Sie sind damit einerseits auf die Windows 8-Plattform beschränkt, andererseits lassen sich auch viele WinRT-typischen Funktionen nicht nutzen.

Doch zurück zu den "echten" WinRT-Apps. Wie Sie obiger Abbildung entnehmen können, lassen sich derzeit zwei Hauptentwicklungsrichtungen erkennen:

- XAML für den Oberflächenentwurf und C++, C# oder VB für die Programmlogik
- HTML 5 für die Oberfläche und JavaScript für die Logik

Beide Lösungen setzen komplett, d.h. mit allen Restriktionen, auf WinRT auf, im Fall von C# und VB.NET kommt noch eine kleine Teilmenge des .NET-Frameworks hinzu (diese ist nicht in der Abbildung enthalten)[1].

HINWEIS: Beachten Sie auch, dass bei WinRT-Apps keinerlei direkter Zugriff auf die Windows API möglich ist, im Gegenteil: selbst viele WinRT-Funktionen sind aus Sicherheitsgründen für die App nur nutzbar, wenn der App die erforderlichen Rechte erteilt wurden (durch den Anwender).

17.4.2 Die WinRT-API

Bisher haben wir nur recht allgemein von der WinRT-API gesprochen, doch was verbirgt sich im Einzelnen dahinter?

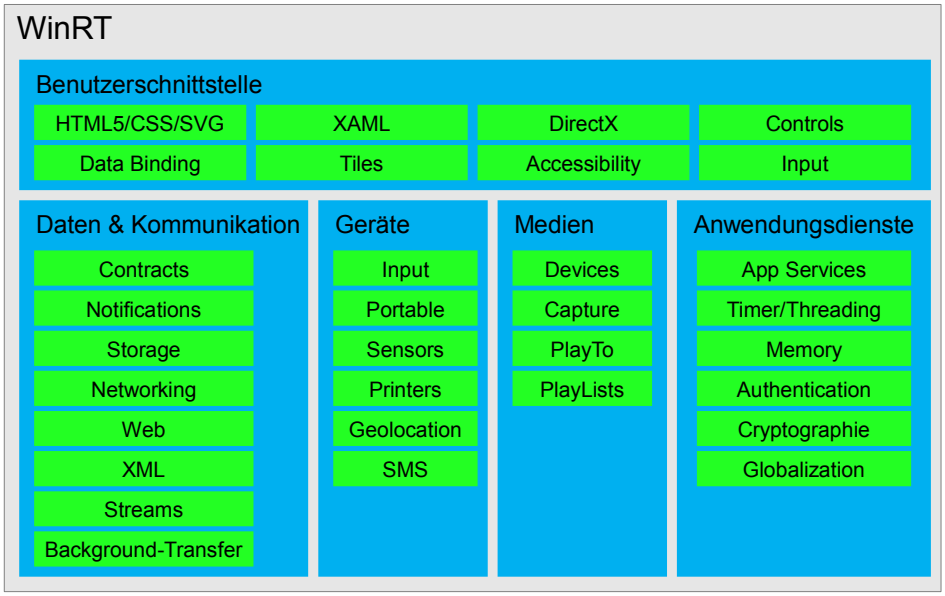
Die Aufgabengebiete und Dienste von WinRT lassen sich grob in fünf Bereiche einteilen:

- Benutzerschnittstelle
- Daten und Kommunikation
- Geräteunterstützung
- Medienunterstützung
- grundlegende Anwendungsdienste

Die folgende Abbildung zeigt etwas detaillierter, welche Funktionen sich hinter den einzelnen Bereichen verstecken.

HINWEIS: Im Interesse des besseren Verständnisses haben wir auf eine Eindeutschung der einzelnen Funktionen verzichtet, als Programmierer dürften Ihnen die obigen Bezeichnungen weitgehend geläufig sein, auch die Analogie zu den entsprechenden Namensräumen lässt sich so besser herstellen.

[1] Naturgemäß beschränken wir uns in diesem Buch auf die XAML/C#-Variante.



Benutzerschnittstelle

Hierbei handelt es sich um alle Aufgaben, die im Zusammenhang mit der Darstellung der WinRT-Apps anfallen. Dies sind natürlich vor allem die Schnittstellen zu den WinRT-Controls, die Sie als C#-Programmierer per XAML zu entsprechenden Oberflächen (Pages) zusammenstellen[1]. Da WinRT im Gegensatz zu WPF nicht auf GDI sondern auf DirectX aufsetzt, sind auch diese Funktionen enthalten.

WinRT stellt auch die Schnittstellen für die Definition der Tile (die Kacheln des WinRT-Desktops) Ihrer Anwendung bereit (dies ist der Ersatz für Desktop-Links und Startmenü-Einträge).

Daten & Kommunikation

Hier bietet WinRT eine Abstraktionsschicht für die Kommunikation zwischen den Anwendungen und dem System. An dieser Stelle werden Sie als Programmierer vermutlich die größten Umstiegsprobleme haben, unterscheidet sich doch dieses Modell grundlegend von den bisherigen Desktop-Anwendungen. Der Datenaustausch mit anderen Anwendungen und auch dem System wird nunmehr über Contracts realisiert, eine direkte Interprozesskommunikation ist nicht mehr möglich.

Zentrale Notifications zeigen Meldungen an oder ermöglichen eine Aktualisierung der App-Tile. Wohlgemerkt – dafür ist WinRT zuständig, nicht Ihre Anwendung.

Vor allem das Storage-Interface wird Sie beim Umstellen Ihrer Anwendungen mit einigen Problemen konfrontieren. Bisher hatten Sie ja recht frei über die Festplatten des Endanwenders verfügen können. Dies ist nun nicht mehr möglich. WinRT kapselt Ihre Anwendung weitgehend von der Umgebung ab, Sie dürfen nur noch auf per Contract freigegebene Verzeichnisse zugreifen bzw.

[1] Die HTML5+SVG-Unterstützung ist vor allem für die JavaScript-Programmierer von Interesse.

müssen explizit einen zentralen Dateiauswahl-Dialog verwenden, um den Nutzer in das Speichern einer Datei mit einzubeziehen.

Geräte und Medien

Eine der erfreulichsten Neuerungen von WinRT: Im Gegensatz zur alten Windows-API verfügen Sie jetzt als Programmierer endlich über eine objektorientierte Schnittstelle zu den einzelnen Sensoren und Geräten die auf dem Anwender-PC bzw. -Tablet verfügbar sind (Lage-, GPS-Sensoren, Webcam etc.)[1].

Anwendungsdienste

Hier finden sich die grundlegenden Funktionen zum Installieren und Aktualisieren der App per Windows Store, die Umgebung für den Start und das Beenden bzw. Anhalten (Suspend) der App und die damit verbundene Speicherverwaltung und Sicherheitsfunktionen.

HINWEIS: Wie die einzelnen APIs im Zusammenhang genutzt werden ist Gegenstand der beiden folgenden Kapitel. Wir denken, dass eine Einführung an Hand praktischer Beispiele sinnvoller ist, als ausschweifende theoretische Erörterungen.

17.4.3 Wichtige WinRT-Namespaces

Im Folgenden finden Sie eine Liste der grundlegendsten Namespaces in WinRT:

```
Windows.ApplicationModel
Windows.Data
Windows.Devices
Windows.Foundation
Windows.Globalization
Windows.Graphics
Windows.Management
Windows.Media
Windows.Networking
Windows.Security
Windows.Storage
Windows.System
Windows.UI[
Windows.UI.Xaml
Windows.Web
```

Wie Sie schnell erkennen können, sind alle WinRT-Namespaces jetzt unter der Wurzel *Windows* zu finden, was die Identifikation und Trennung von den bisherigen Namespaces erleichtert

[1] Wer schon mal auf eine Webcam, ein Mikrofon oder einen Scanner zugreifen wollte, kann ein Lied davon singen. Ein buntes Gemisch aus API, MCI, WIA, TWAIN, DirectX etc. machte das Leben schwer.

HINWEIS: Dies sind nur die Haupt-WinRT-Namespaces, nicht die Namespaces, die Ihnen im Rahmen des .NET WinRT-Profils zur Verfügung stehen.

Neben den WinRT-Namespaces stehen Ihnen in C# auch noch folgende grundlegende .NET-Framework-Namespaces zur Verfügung:

```
System
System.CodeDom.Compiler
System.Collections
System.ComponentModel
System.Diagnostics
System.Dynamic
System.Globalization
System.IO
System.Linq
System.Net
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.ServiceModel
System.Text
System.Text.RegularExpressions
System.Threading
System.Threading.Tasks
System.Windows.Input
System.Xml
Microsoft.CSharp.RuntimeBinder
Microsoft.VisualBasic
```

HINWEIS: Welche Unter-Namespaces im Einzelnen verfügbar sind, ist schnell ersichtlich, wenn Sie ein WinRT-App-Projekt bearbeiten. Hier werden automatisch alle nicht verfügbaren Namespaces und deren Klassen ausgeblendet.

17.4.4 Der Unterbau

Wir hatten es Ihnen ja schon verraten, die Windows Runtime basiert auf der guten alten COM-Technologie (*Component Object Model*), hat allerdings einige wichtige Fähigkeiten neu dazugewonnen.

Die Schnittstellen

Für Sie als C#-Programmierer ist dieser Unterbau eigentlich von untergeordnetem Interesse, da Sie nicht, wie z.B. bei der OLE-Automation, direkt mit den COM-Objekten arbeiten müssen. Es schadet jedoch nicht zu wissen, dass die unterste WinRT-Ebene auf Interfaces statt auf Objekten

basiert, die in jedem Fall zunächst die Schnittstellen *IUnknown* und *IInspectable* unterstützen. Letztere dient der dynamischen Bindung mit JavaScript, erstere ist für die .NET-Sprachen relevant.

Über diese beiden Schnittstellen können die weiteren vom Objekt implementierten Interfaces abgefragt werden. Dieses Vorgehen hat den Vorteil, dass das ursprüngliche Objekt ausgetauscht werden kann, ohne dass der darauf basierende Code neu zu kompilieren wäre. Wichtig ist nur, dass sich das verwendete Interface nicht ändert (quasi die Schnittstelle zu Ihrem Programm bzw. zu Ihrer App).

Language Projection

Von all dem bekommen Sie als C#-Entwickler jedoch nichts mit. Dank der neu eingeführten umfangreichen Unterstützung für Metadaten (hier hat COM von .NET gelernt) und der Unterstützung für Language Projections (siehe Abschnitt 17.1.6) kann Ihnen der jeweilige Compiler und die Visual Studio IDE (IntelliSense) viel Arbeit abnehmen und die "schmutzigen" Details vor Ihren Augen verbergen. Dazu wird vom Compiler die rein interface-basierte WinRT-Schnittstelle in C#-typische Klassen umgewandelt. Dies betrifft jedoch nicht nur die Klassen, sondern auch alle anderen Sprachkonstrukte wie Primitive, Enums, Structs, Delegates sowie die Klassenmember (Konstruktoren, Methoden, Eigenschaften, Ereignisse).

Einige spezielle WinRT-Collection-Interfaces werden auf die bekannten .NET-Interfaces gemappt:

WinRT	.NET
IVector<T>	*IList<T>*
IVectorView<T>	*IReadOnlyList<T>*
IIterable<T>	*IEnumerable<T>*
IMap<Key,Value>	*IDictionary<Key,Value>*
IMapView<Key,Value>	*IReadOnlyDictionary<Key,Value>*

Doch wie hat man sich den internen Ablauf eigentlich vorzustellen? Ein einfaches Beispiel für das Erzeugen einer Instanz soll die prinzipielle Vorgehensweise verdeutlichen:

- Aufruf *new* in Ihrem Programm
- mittels Projection wird der Name der Klasse an *RoActivateInstance()* übergeben
- das entsprechende Package wird im Katalog gesucht (Registry[1])
- die DLL wird geladen
- über mehrere Zwischenschritte wird das grundlegende COM-Objekt erstellt
- mittels *IUnknown*- oder *IInspectable*-Interface wird per Projection ein Wrapper erzeugt
- der Wrapper wird an das COM-Objekt gebunden
- der Wrapper wird an die App übergeben

[1] Das Ende der Registry wurde zwar von diversen Gurus schon vor einiger Zeit verkündet, aber Totgesagte leben bekanntlich länger!

Wie bereits gesagt, von all dem bekommen Sie nichts mit, aus Ihrer Sicht arbeiten Sie mit ganz gewöhnlichen Klassen.

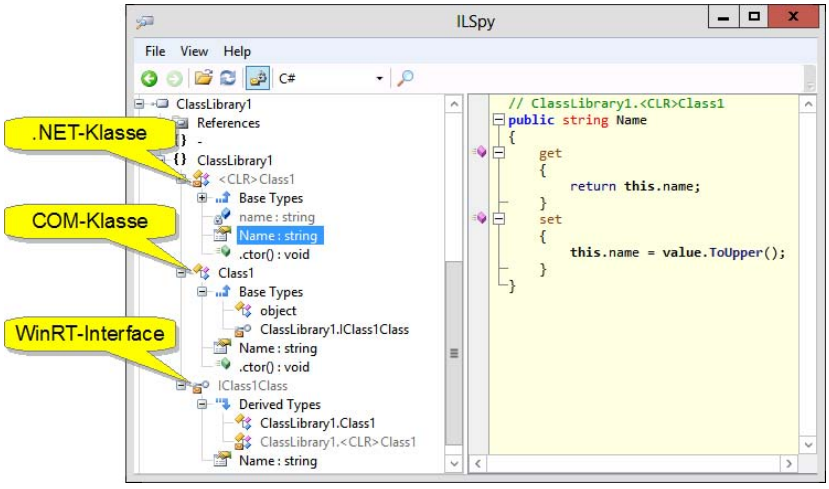
Assemblies für WinRT

Sollten Sie in die Verlegenheit kommen, eigene Assemblies für WinRT zu entwerfen, die vielleicht zusammen mit einem C++ oder JavaScript-Programm laufen sollen, müssen Sie sich jedoch etwas detaillierter mit dieser Materie beschäftigen. In diesem Fall sind Sie an einige Restriktionen gebunden:

- Öffentliche Klassen müssen *sealed* oder *static* sein.
- Öffentliche *structs* dürfen keine Methoden, Ereignisse, Konstruktoren definieren.
- Alle öffentlichen Felder, Parameter, Eigenschaften etc. müssen sich auf die von WinRT bekannten Typen beschränken.
- Sie können nur systemkonforme Generics verwenden.

HINWEIS: Wohlgemerkt, dies gilt alles nur für die öffentliche Schnittstelle, was Sie intern in Ihrer Assembly anstellen ist vollkommen egal (an die WinRT-Spielregeln müssen Sie sich aber trotzdem halten).

Der Entwurf selbst ist recht simpel: Sie erstellen in Visual Studio zunächst eine Klassenbibliothek und setzen nachfolgend unter *Projekt|Eigenschaften|Anwendung* den *Ausgabetyp* auf *WinMD-Datei*. Erstellen Sie jetzt das Projekt, wird statt der DLL eine WinMD-Datei erzeugt. Im Gegensatz zu den systemeigenen WinMD-Dateien enthält unsere Datei neben den für WinRT erforderlichen Interfaces auch den eigentlichen Code, wie es ein Blick in die Assembly verrät:



Mehr zu WinRT finden Sie in den folgenden Kapiteln, wir aber machen vorher noch einen kleinen Exkurs.

17.5 Gedanken zum Thema "WinRT & Tablets"

Bevor wir uns weiter in die WinRT-Entwicklung vertiefen, möchten wir Sie noch mit einigen Gedanken konfrontieren, die uns als Autoren im Zusammenhang mit der WinRT-Einführung so durch den Kopf gegangen sind.

17.5.1 Windows 8-Oberfläche versus Desktop

Man kann ja beim Thema "Oberflächendesign" verschiedener Meinung sein, aber der Versuch, die Windows 8-Oberfläche (Modern UI) auf Desktop-PCs zu etablieren, ist schon recht "mutig".

Ist die Windows 8-Oberfläche auf einem Notebook/Tablet vielleicht sinnvoll, stellt sich bei 24" bzw. 27" Bildschirmdiagonale wohl eher "Augenkrebs" ein. Eine Multimonitorlösung mit Modern UI und Desktop gleichzeitig ist wohl auch nicht "das Gelbe vom Ei". Riesige Kacheln verbrauchen viel Platz und bieten an einem normalen Arbeitsplatz keinerlei Vorteil. Im Gegenteil: die Maus ist dauernd von einer Ecke des Bildschirms zur anderen unterwegs, eine Übersicht stellt sich nicht ein. Das nervige Geflacker nach dem Aufruf einer Nicht-WinRT-App (Anzeige Desktop + Anwendung) und wieder zurück (beim Beenden) geht auch auf die Nerven.

Nein – wir brauchen auf einem Arbeitsplatz-PC nicht andauernd neu aufflackernde Apps, die uns mit Informationen belästigen, die wir nicht haben wollen. Die gute alte Taskbar-Notification-Area reicht völlig aus: immer gut im Blickfeld und doch nicht nervig.

Nachdem Experimente mit Active Desktop (Windows 98SE), Sidebar (Windows Vista) oder Gadgets (Windows 7) fehlgeschlagen sind[1], erfolgt jetzt also der Frontalangriff auf den gesamten Desktop per Windows 8-UI.

Einziger Grund für diese Innovation: Der Nutzer soll mit leichtem Druck zu Apps "überredet" werden, und diese lassen sich rein zufällig nur über den Microsoft-Shop erwerben. Dort wiederum verdient Microsoft bei jedem Kauf automatisch mit.

17.5.2 Tablets und Touchscreens

Ja, ja, die Touch-Fähigkeiten der neuen Oberfläche – angefeuert durch den derzeitigen Tablet-Hype[2] will natürlich auch Microsoft auf diesem Gebiet nicht zurückstehen. An einem Desktop-Arbeitsplatz wird wohl kaum einer, der Verstand hat, einen Touchscreen regulär verwenden. Bleiben die so genannten Tablets. Doch mal ganz nüchtern betrachtet – wer eigentlich soll die Tablets wofür genau nutzen? Dazu einige Gedanken, bevor hier einigen "die Pferde durchgehen".

- **Tablet-Verwendung**
  Spiele, Spiele, Spiele … dann eine ganze Weile nichts und dann Soziale Netzwerke, E-Mail, Browser, E-Books, Bilder und Videos ansehen und … tja was eigentlich noch Sinnvolles? Für die ersten Punkte reicht auch ein Smartphone. Wollen Sie vielleicht einen Brief auf dem Tablet schreiben? Bildbearbeitung? Na ja, das dürfte wohl eher ein Wunschtraum bleiben.

[1] Nutzen Sie eines dieses Features?

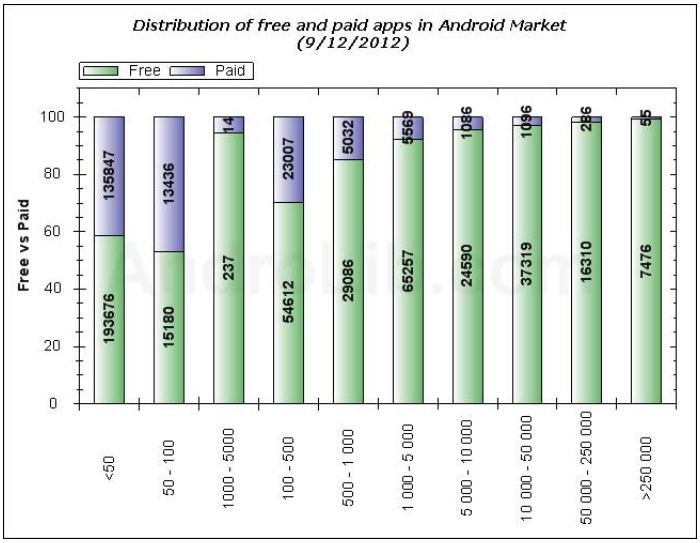
[2] Es wird ja schon das Ende des PCs verkündet – was für ein Unsinn!

- **Tablet-Hardware**
  Wer hier an die Übernahme seiner Windows-Anwendungen denkt – vergessen Sie es! Entscheiden Sie sich für ein Intel-Tablet, können Sie auch gleich ein günstigeres Notebook mit besserer Technik und vor allem längerer Akku-Laufzeit kaufen. Auf diesem nutzen Sie den Touch-Screen garantiert nicht. Kaufen Sie ein ARM-Tablet, sind Sie auf reine WinRT-Anwendungen festgelegt, alle "alten" x86-Anwendungen laufen dort nicht. Die angekündigte Office-Suite für Windows RT ist sicher für die Anzeige von Word- und Excel-Dokumenten ganz nett, aber effektiv arbeiten wird damit wohl kaum einer.

- **Zielgruppe**
  Ganz nebenbei sollte man auch mal den Käufer im Auge behalten. Die Apple iPad-Nutzer werden wohl kaum in Scharen zu Windows überlaufen, dazu ist der "Angeber-/Lifestyle-Faktor" beim iPad einfach höher, abgesehen davon, dass es sich um ein recht ausgereiftes Produkt handelt. Android-Nutzer haben Ihre Plattform gewählt, weil Sie eben nicht von einem Shop abhängig sein wollen und weil Sie eine offene Plattform bevorzugen. Bleibt der bisherige Windows-PC-Nutzer, der ein Zweitgerät (Freizeit) erwirbt. Doch an dieser Stelle stellt sich die Frage, warum diese Zielgruppe nicht schon ein Tablet besitzt, wenn Sie denn eins bräuchte.

- **Geräte-Konkurrenz**
  Wenn Sie die Wahl zwischen PC, Smartphone und Tablet haben, auf welches Gerät können Sie problemlos verzichten? Die Antwort dürfte klar sein.

- **Mit Apps "Geld ohne Ende" verdienen**
  Da träumen Sie mal schön weiter. Ein paar Zahlen gefällig? Der iPad-Marktanteil liegt bei zirka 60% (diese Zielgruppe fällt aus), der von Android bei 30% (Überläufer gering). Anwendungen im iTunes App Store (Apple) bzw. im Android-Market (Google): jeweils zirka 650.000 (das ist Ihre Konkurrenz). Aufschlussreich ist auch folgende Auswertung der Verkaufszahlen aus dem Android-Market (Quelle: *http://www.androlib.com/appstats.aspx* vom 12.9.2012):



Wer hier tatsächlich Geld verdienen will, muss bei den aktuellen App-Preisen schon mehr als 50.000 Verkäufe haben[1]. Auch ja, in der obigen Grafik sind natürlich auch noch die Verkäufe für ganz normale Smartphones mit dabei (nicht Ihre Zielgruppe mit WinRT). Da wird die Luft schon sehr viel dünner, wenn es um die wolkig versprochenen Zielgruppen und den Geldregen geht. Zum Thema "Zahlungsbereitschaft" im Internet noch folgende Zahlen: Die iPad-Nutzer geben täglich ca. 70.000 Dollar für den Newsstand[2] aus. Hört sich gut an, ist es aber nicht. Das sind umgerechnet 35.400 Euro (nach Abzug der Apple-Umsatzbeteiligung) bei rund 60 Millionen verkaufter iPads. Die durchschnittliche Zahlungsbereitschaft pro Gerätenutzer können Sie sich ja selbst ausrechnen. Damit würde keine Lokalzeitung klarkommen und das bei einer weltweiten Zielgruppe. Eine zweite Zahl: 89% aller heruntergeladenen Smartphone-Apps sind kostenlos, dieser Anteil steigt von Jahr zu Jahr.

HINWEIS: Wir wollten Ihnen natürlich nicht die Lust am Programmieren nehmen, sondern Sie lediglich zum Nachdenken anregen und vor allzu großen Illusionen bewahren.

17.6 Praxisbeispiel

17.6.1 WinRT in Desktop-Applikationen nutzen

Wie schon mehrfach erwähnt, lassen sich die neuen WinRT-Klassen auch in "konventionellen" Desktop-Applikationen (Windows Forms, WPF) oder auch direkt in Konsolenanwendungen einsetzen. Allerdings sind Sie an einige Restriktionen gebunden, dies betrifft vor allem

- die Zielplattform (nur Windows 8),
- keine Verwendung von WinRT-Komponenten

Da es sich im Folgenden nur um eine kleine Demo handelt, wollen wir uns auf das einfache Auslesen der Gerätekonfiguration mittels WinRT und die Anzeige in einer *Treeview* beschränken.

HINWEIS: Sie müssen das Projekt unter Windows 8 erstellen!

Oberfläche

Erstellen Sie zunächst ein neues WPF-Projekt und erweitern Sie die Definition von *MainWindow.xaml* um folgende Zeilen:

```
<Window x:Class="WPF_WinRT.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF + WinRT" Height="350" Width="525">
    <DockPanel>
```

[1] Vergessen Sie nicht: der Shop-Betreiber bekommt noch 30% und das Finanzamt will auch nicht leer ausgehen.

[2] Zeitungskiosk

```
        <Button DockPanel.Dock="Top" Content="Start" Height="40" Click="Button_Click_1"/>
        <TreeView Name="TreeView1"/>
    </DockPanel>
</Window>
```

Wir zeigen lediglich eine Schaltfläche und eine *TreeView* an, die Ausrichtung der Komponenten überlassen wir dem *DockPanel*.

Quellcode

Bevor Sie sich in die WinRT-Programmierung stürzen können, müssen Sie noch die drei folgenden Verweise in das Projekt aufnehmen:

- *System.Runtime*
- *System.Runtime.WindowsRuntime.dll*
- *Windows*

Die ersten beiden Verweise müssen Sie im Verweisdialog per *Browse*-Schaltfläche suchen:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\...
```

Um den *Windows*-Verweis einzubinden, editieren Sie die Projektdatei **.csproj* und fügen den folgenden Eintrag für die Zielplattform hinzu:

```
...
    <AssemblyName>WPF_WinRT</AssemblyName>
    <TargetPlatformVersion>8.0</TargetPlatformVersion>
    <TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
...
```

Nachfolgend können Sie im Verweisdialog über *Windows|Kern* die *windows.winmd*-Datei auswählen.

Nun endlich können wir uns dem eigentlichen Code zuwenden:

```
...
using Windows.Devices.Enumeration;

namespace WPF_WinRT
{
    public partial class MainWindow : Window
    { ...
```

Mit dem Klick auf die Schaltfläche soll die *TreeView* gefüllt werden. Da der WinRT-Aufruf asynchron ist, müssen wir die Methode mit *async* kennzeichnen:

```
        async private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Hier folgt der eigentliche WinRT-Aufruf:

```
            DeviceInformationCollection dicol = await DeviceInformation.FindAllAsync();
```

Wie Sie sehen, müssen Sie sich nicht mit komplizierten Umwandlungen beschäftigen, die Verwendung entspricht der einer normalen .NET-Klasse. Die Auswertung der Collection bzw. das Füllen der *TreeView* übernehmen die beiden folgenden *foreach*-Anweisungen:

```
            foreach (DeviceInformation di in dicol.OrderBy(di => di.Name))
            {
```

Der jeweilige Haupteintrag:

```
                TreeViewItem tvi = new TreeViewItem { Header = di.Name };
```

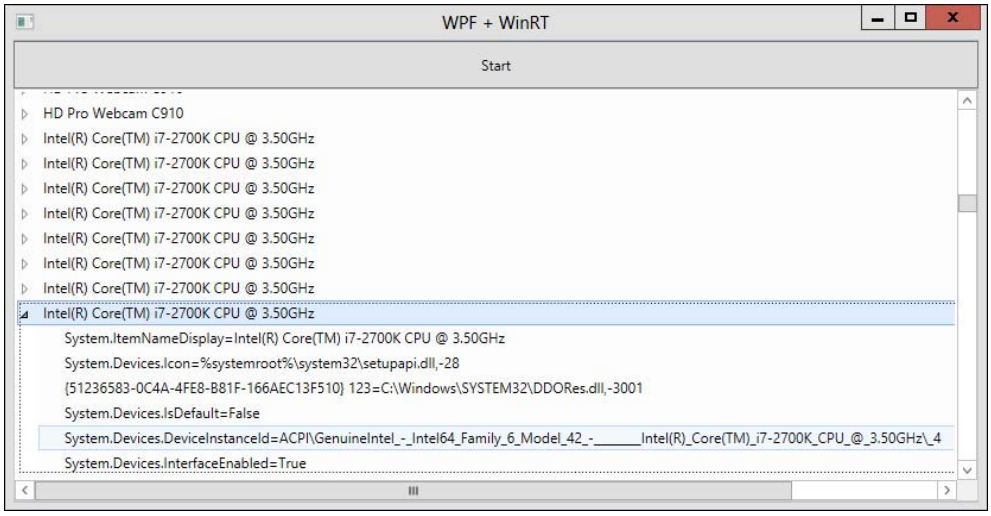
Hier lesen wir noch die Eigenschaften aus, diese befinden sich in einer extra Collection:

```
                foreach (var prop in di.Properties)
                    tvi.Items.Add(new TreeViewItem { Header = prop.Key + "=" + prop.Value });
                TreeView1.Items.Add(tvi);
            }
        }
    }
}
```

Wie Sie sehen, die Verwendung der WinRT-Klassen ist ohne großen Aufwand möglich.

Test

Nach dem Start sollte sich die Baumansicht füllen, je nach Konfiguration kann es etwas dauern:



Bemerkung

Ohne die Möglichkeiten von WinRT müssten Sie sich näher mit den WMI (Windows Management Instrumentation) beschäftigen, einer Schnittstelle zur Abfrage von Hard- und Softwareinformationen. Allerdings sollten Sie sich dann auch mit der Abfragesprache WQL (*WMI Query Language*) auskennen, die stark an SQL angelehnt ist.

Kapitel 18

WinRT-Oberflächen entwerfen

Windows 8 ist zunächst einmal Oberfläche pur. Aus diesem Grund wollen wir uns in einem ersten Schritt mit den Entwurfskonzepten und den wichtigsten Elementen für WinRT-Oberflächen beschäftigen, bevor wir uns um Logik, Datenaustausch und weitere Spezialitäten kümmern.

HINWEIS: Es wäre sehr sinnvoll, wenn Sie bereits mit den Themen XAML, WPF und WPF-Datenbindung vertraut sind. Fast alle dieser Konzepte lassen sich auch auf die WinRT-Programmierung übertragen. Um uns nicht endlos zu wiederholen, verweisen wir deshalb auf die einführenden Kapitel 12 bis 15.

18.1 Grundkonzepte

Als gestandener Programmierer sitzen Sie nun schon zum wiederholten Mal auf der "Schulbank". Viele von Ihnen haben die Evolution von Windows Forms über WPF/XAML zu Silverlight mitgemacht und fragen sich jetzt sicher, was kann ich davon weiterhin nutzen?

Eine Beruhigungspille vorweg: Wenn Sie mit den WPF- bzw. besser noch Silverlight-Techniken vertraut sind, können Sie fast problemlos Ihre bisherigen Kenntnisse weiter verwenden. Windows Forms-Programmierer erwartet jedoch zunächst ein kompletter Kulturschock: Angefangen beim Programmieren der Oberflächen (jetzt per XAML-Code[1]) über die Konzepte zur Datenbindung – fast alles ist neu und muss zunächst eingeordnet und erlernt werden. Ein Trost: Ihre C#-Kenntnisse müssen Sie zwar an der einen oder anderen Stelle auffrischen, aber im Grunde können Sie diese weiter nutzen.

Doch wo fangen wir bei so gravierenden Änderungen am besten an?

Wir ignorieren zunächst die vielen Projektdateien, die Sie schon im vorhergehenden Einführungsbeispiel (Seite 876) bewundern durften und wenden uns ganz schlicht der Programmoberfläche und den Oberflächenkonzepten von WinRT zu. Aufbauend darauf werden wir die Startroutinen etc. näher unter die Lupe nehmen. Der Vorteil: Die Lernkurve ist nicht ganz so steil (wir nutzen quasi

[1] Wir schreiben bewusst per "Code", da Sie mit dem XAML-Editor wesentlich schneller und intuitiver arbeiten können als mit dem grafischen Designer. Dies betrifft vor allem die Organisation der Seiten über die Layout-Controls.

Serpentinen) und Sie bekommen auch gleich etwas zu sehen, statt endlos mit Hintergrundroutinen, Kontrakten etc. belästigt zu werden. Wundern Sie sich also nicht, wenn Ihre neuen WinRT-Programme noch nicht nicht allzu viel Logik aufweisen.

18.1.1 XAML (oder HTML 5) für die Oberfläche

Sie haben es ja nun schon mehrfach gelesen: die Oberfläche Ihrer WinRT-Apps basiert auf der Beschreibungssprache XAML (*eXtensible Application Markup Language*, gesprochen "Xemmel"), sofern Sie die Anwendungen mit C# oder VB.NET entwerfen. Ganz anders die JavaScript-Programmierer, diese nutzen stattdessen HTML 5. Da Sie ein C#-Buch in den Händen halten, können wir uns jedoch voll und ganz auf XAML beschränken.

Die Verwendung von XAML bedeutet:

- eine strikte Trennung von Oberfläche und Code[1]
- maximale Flexibilität bei der Gestaltung von Oberflächen
- die Fähigkeit zum leichten Anpassen der Oberfläche per Designs
- leichterer Entwurf von dynamischen Oberflächen basierend auf Layout-Controls

Doch wo viel Licht, da auch viel Schatten: Sie werden sehen, dass auch schon kleine Oberflächen recht lange XAML-Quellcodes nach sich ziehen. Als Windows Forms-Programmierer blieben Sie ja davon verschont, all die endlosen Eigenschaftszuweisungen etc. fanden sich in den etwas versteckten **.Designer.cs*-Dateien.

Sehen wir uns also eine dieser XAML-Dateien näher an.

BEISPIEL 18.1: Die Visual Studio-Vorlage "Leere Seite" erzeugt folgenden XAML-Code

XAML

Das Root-Element ist *Page*:

```
<Page
```

Hier ist die Verknüpfung zum entsprechenden Klassen-Code zu finden:

```
    x:Class="Vorlage.BlankPage1"
```

Hier werden die Namespaces definiert, die vom XAML-Code genutzt werden können:

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

Dieser Namespace ist besonders wichtig, handelt es sich doch um einen Verweis auf den aktuellen Projektnamespace:

```
    xmlns:local="using:Vorlage"
    mc:Ignorable="d">
```

[1] So weit die Theorie. Die Profis wissen jedoch, dass es auch möglich ist, Code-Objekte im XAML-Code zu instanziieren, was die heile Welt gleich wieder durcheinanderbringt.

BEISPIEL 18.1: Die Visual Studio-Vorlage "Leere Seite" erzeugt folgenden XAML-Code

WPF-Programmierer kennen hier eine andere Syntax, diese dürfte jedoch einfacher sein.

Da *Page* nur ein Element als Content enthalten kann, benötigen wir eines der so genannten Container-Controls, in diesem Fall ein *Grid*, in dem wiederum beliebig viele Controls angeordnet werden können:

```
    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

Hier kann Ihr XAML-Code stehen!

    </Grid>
</Page>
```

18.1.2 Die Page, der Frame und das Window

An dieser Stelle werden die WPF-Programmierer vielleicht stutzen. Warum wird in obigem Beispiel eigentlich *Page* und nicht *Window* als Root-Element verwendet? Die Frage ist nach einem Blick in die Datei *App.xaml.cs* schnell beantwortet, denn hier findet sich der schon vorbereitete Eintrittspunkt in unser Programm:

BEISPIEL 18.2: Eintrittspunkt in *App.xaml.cs*

C#
```
...
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
            Frame rootFrame = Window.Current.Content as Frame;
            if (rootFrame == null)
            {
                rootFrame = new Frame();
                if (args.PreviousExecutionState == ApplicationExecutionState.Terminated) { }
                Window.Current.Content = rootFrame;
            }
            if (rootFrame.Content == null)
            {
                if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
                  throw new Exception("Failed to create initial page");
            }
            Window.Current.Activate();
        }
...
```

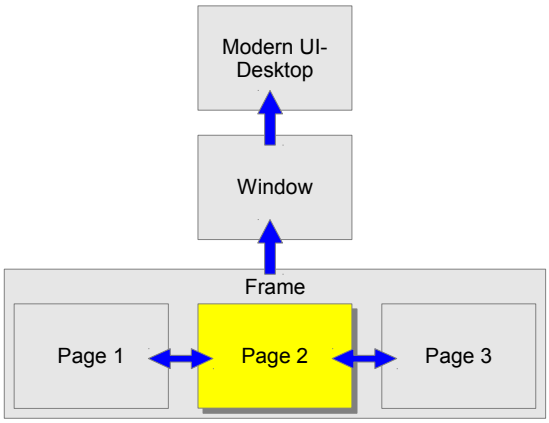
Folgendes passiert:

1. Es wird ein vorhandenes *Frame*-Objekt abgerufen oder ein neues erzeugt.
2. Der statischen *Window*-Klasse wird das *Frame*-Objekt zugewiesen.
3. Dem Frame wird die Klasse *MainPage* zugewiesen.
4. Das Window wird aktiviert, d h. in den Vordergrund gebracht und angezeigt.

Der Grund für diesen "Handstand":

- Jede WinRT-Anwendung kann nur **ein Window** enthalten (das ist obige statische Klasse)! Dies entspricht einem Webbrowser in Vollansicht.
- In diesem Window sollen jedoch wechselnde Inhalte angezeigt werden. Das sind die Pages, ähnlich wie die Seiten im Webbrowser.
- Für die Organisation (Übergänge, Navigation) zwischen den Pages sorgt der Frame (quasi die Navigationstasten des Webbrowsers).

Die folgende Abbildung zeigt noch einmal den Sachverhalt:



Wenn Sie dieses Konzept verstanden haben, wird Ihnen auch klar, warum folgender Code problemlos funktioniert:

```
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
...
            Window.Current.Content = new Button() { Content = "Hallo World" };
            Window.Current.Activate();
        }
```

In diesem Fall wird ein schwarzer Bildschirm mit einer Schaltfläche "Hello World" angezeigt, da der Button das einzige mögliche Element unseres Windows ist. Ein Wechsel zu anderen Ansichten ist jetzt nicht möglich, es sei denn, Sie tauschen den Button per Code gegen ein anderes Control aus.

18.1.3 Das Befehlsdesign

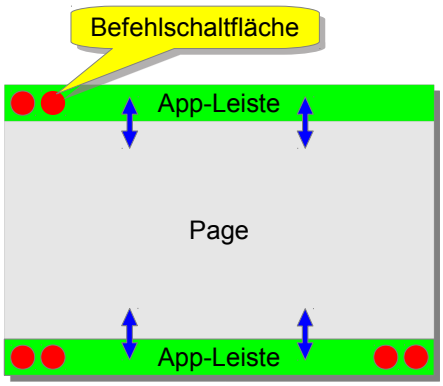
Bevor Sie jetzt anfangen, frei nach MUPS[1] Ihre Oberflächen zu entwerfen, sollten Sie sich zunächst mit einigen grundlegenden Design-Richtlinien für die Gestaltung der WinRT-Oberflächen vertraut machen:

[1] Methode des Unbekümmerten Probierens

HINWEIS: Die App-Oberfläche sollte sich grundsätzlich auf die wesentlichsten Gestaltungselemente beschränken, der Content/die Information steht im Mittelpunkt.

App-Leisten

Für optionale Steuerelemente (Konfiguration, Navigation etc.) stehen Ihnen die App-Leisten am oberen und unteren Bildschirmrand zur Verfügung. Diese werden optional eingeblendet und im Normalfall automatisch ausgeblendet, wenn der Nutzer eine Aktion ausgewählt hat. Die App-Leisten werden entweder per Code, per rechter Maustaste oder mittels Wischbewegung der Finger angezeigt.



Die folgende Abbildung zeigt die Wetter-App mit den beiden eingeblendeten App-Leisten am oberen und unteren Bildschirmrand:

Wie Sie sehen, sind die enthaltenen Befehlsschaltflächen nicht irgendwo angeordnet, sondern an den jeweiligen Außenseiten. Der Grund liegt in der Bedienung der Oberfläche, wenn es sich um ein touch-fähiges Gerät handelt: Sie halten dieses meist mit beiden Händen und bedienen mit den Daumen. Wer allerdings diese Oberfläche mit einer Maus bedienen "darf", ist sicher nicht glücklich

darüber. Von einer zur anderen Befehlsschaltfläche sind es auf einem 26" oder 30" Monitor gefühlt mehrere "Kilometer" – der Spagat zwischen zwei grundverschiedenen Ausgabegeräten ist nun mal nicht so einfach zu überbrücken, wie es sich die Microsoft-Entwickler gedacht haben.

Kommen Sie jetzt bitte nicht auf die Idee, derartige Leisten am linken oder rechten Rand einzublenden, dort sollte Platz für die WinRT-eigenen-Leisten bleiben. Derartige "Features" würden den Anwender nur verwirren.

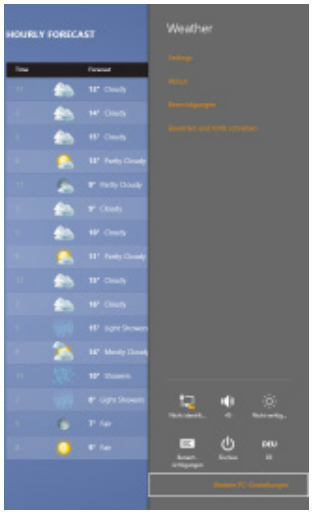
Weitere Informationen finden Sie unter der folgenden Adresse, Microsoft hat sich da schon reichlich Gedanken zum Thema gemacht:

LINK: `http://msdn.microsoft.com/de-de/library/windows/apps/hh761499`

HINWEIS: Wie Sie die Leisten (*ApplicationBar*) in Ihren WinRT-Apps selbst erzeugen und nutzen, zeigt Abschnitt 21.3.5 ab Seite 1103.

Settingsbereich

Muss Ihr Programm umfangreichere Informationen zur Konfiguration etc. zur Verfügung stellen, können Sie sich auch in den Settingsbereich der WinRT-Oberfläche "einklinken". Dieser Bereich wird über die Windows-Taste+I oder eine Wischbewegung am rechten Rand angezeigt:



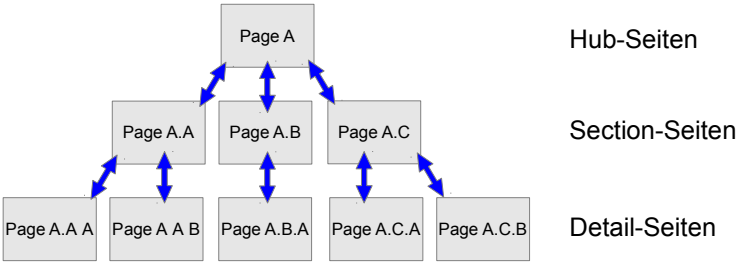
Wie Sie in obiger Abbildung sehen, hat die Wetter-App vier Schaltflächen in den Settingsbereich eingefügt, die die Navigation zu extra Seiten oder Dialogen im Settingsbereich ermöglichen.

18.1.4 Die Navigationsdesigns

Grundsätzlich sollten Sie sich schon vor dem Erstellen der Anwendung darüber im Klaren sein, welches Navigationsdesign Sie verwenden. Formular-orientierte Programmierer müssen hier etwas umdenken oder dazu lernen, ASP.NET-Programmierer wissen sicher schon mehr.

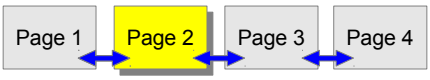
Da wir nur eine Seite innerhalb unserer App darstellen können, müssen wir uns klar darüber werden, in welcher Reihenfolge zwischen den Seiten gewechselt werden kann. Dabei bieten sich zwei grundsätzliche Varianten an:

- Hierarchische Organisation



  Diese entspricht dem Aufbau der meisten Nachrichten-Webseiten, von einer Startseite werden Sie meist zu den Rubriken und von dort zu den einzelnen Artikeln weitergeleitet.

- Flache Organisation



  Diese Variante ist mit der Navigation innerhalb von Blogs oder News-Listen vergleichbar. Optional ist der Sprung auf die erste Seite denkbar.

Je nach Organisationstyp wählen Sie später den Projekttyp und die weiteren Detailseiten in Visual Studio aus (siehe Seite 906).

Mehr dazu unter

LINK: hhttp://msdn.microsoft.com/de-de/library/windows/apps/xaml/hh761500.aspx

18.1.5 Achtung: Fingereingabe!

Nach Jahrzehnten der Maus+Tastatur-Programmierung rückt jetzt langsam die Touch- oder Finger-Eingabe in den Fokus der Programmierer. Egal, ob dies in allen Fällen sinnvoll ist oder nicht, gerade die WinRT-Apps sollten in der Lage sein, Eingaben auch per Finger zu verarbeiten. Diese Funktionalität ist zwar schon in den Controls eingebaut, allerdings müssen Sie als Programmierer darauf achten, dass die Controls auch in einem "fingertauglichen" Abstand platziert werden.

Aus diesem Grund finden Sie in den Microsoft-Richtlinien zum Erstellen von WinRT-Apps auch reichlich Informationen über Größe und Abstand von Controls sowie die Verwendung von Gesten:

LINK: http://msdn.microsoft.com/de-de/library/windows/apps/hh465415

Eine weitere Problematik ist die Möglichkeit, nicht nur mit einem Finger eine Aktion auszulösen, sondern mit komplexeren Gesten bestimmte Verhaltensweisen der App zu erzielen.

18.1.6 Verwendung von Schriftarten

Verwenden Sie grundsätzlich so wenig Schriftstile wie möglich, und halten Sie sich an die Vorgaben von Microsoft:

- Standard für Windows 8-Userinterface-Elemente ist *Segoe Light UI*.
- Verwenden Sie *Calibri* für Text den der Anwender liest oder schreibt.
- Größere Textblöcke formatieren Sie am besten mit *Cambria*.

Orientieren Sie sich bei Ihren Oberflächen, sowohl was die Schriftart als auch die Schriftgröße und Position angeht, an den Microsoft-Beispielen und Richtlinien.

- Seitenkopf → *Segoe UI Light 42pt*
- Unterüberschrift → *Segoe UI Light 20pt*
- Normaler Text → *Segoe UI Light 11pt*
- ...

Siehe dazu auch:

LINK: `http://msdn.microsoft.com/de-de/library/windows/apps/hh700394`

18.2 Projekttypen und Seitentemplates

Haben Sie sich mit den Richtlinien vertraut gemacht, kommt sicher schnell die Frage auf, wie aufwändig die Programmierung derartiger Apps ist. Hier können wir Sie trösten, zumindest das erste Grundgerüst lässt sich mittels Visual Studio recht schnell erstellen.

Aktuell stellt Ihnen Visual Studio die Templates

- *Leere App*,
- *Geteilte App* und
- *Raster App*

zur Verfügung. Diese lassen sich als Ausgangspunkt für eigene Experimente nutzen. Alternativ können Sie auch auf die zahlreichen Microsoft-Beispiele aufbauen oder sich ein eigenes Template zusammenstellen, das Ihren Erfordernissen besser entspricht.

18.2.1 Leere App

Wie der Name schon andeutet und wie es auch das Einstiegsbeispiel gezeigt hat, besteht diese App nur aus einer leeren Seite (gleichnamige Vorlage). Allerdings ist das Grundgerüst für eine Seitennavigation bereits vorgesehen, d h., es sind sowohl die Klasse *LayoutAwarePage* (Grundlage der Standardseiten) als auch eine *Frame*-Definition in der *OnLaunched*-Methode implementiert. Sie können in diesem Projekt jederzeit Standardseiten hinzufügen und dann auch zwischen diesen navigieren. Für die entsprechende Navigationslogik müssen Sie in diesem Fall jedoch selbst sor-

gen, die Schaltflächen für den Rücksprung sind bei Verwendung der Vorlage *Standardseite* bereits vorhanden und können automatisch genutzt werden.

Um eine Navigation zwischen zwei Seiten zu realisieren gehen Sie wie folgt vor:

- Erstellen Sie eine neue *Leere App* (Menüpunkt *Datei|Neu|Projekt|Visual C#|Windows Store|Leere App*).
- Fügen Sie dem Projekt eine weitere *Leere Seite* hinzu (Menüpunkt *Projekt|Neues Element hinzufügen|Windows Store|Leere Seite*).
- Fügen Sie in beide Seiten (XAML-Code) eine Schaltfläche hinzu:

```
<Page
...
    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <Button Click="Button_Click_1">Sprung zur Seite 2</Button>
    </Grid>
</Page>
```

  Die Beschriftungen (*Content*) sollten jeweils auf die andere Seite verweisen.

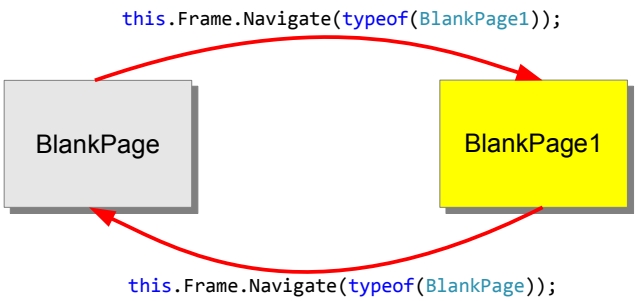
- Definieren Sie folgenden Ereigniscode:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            this.Frame.Navigate(typeof(BlankPage1));
        }
```

  bzw.

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            this.Frame.Navigate(typeof(BlankPage));
        }
```

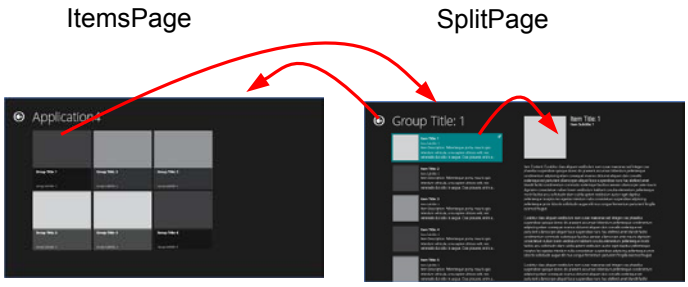
- Starten Sie die App. Sie sollten jetzt die Möglichkeit haben, mit Hilfe der Schaltflächen zwischen beiden Seiten zu wechseln:



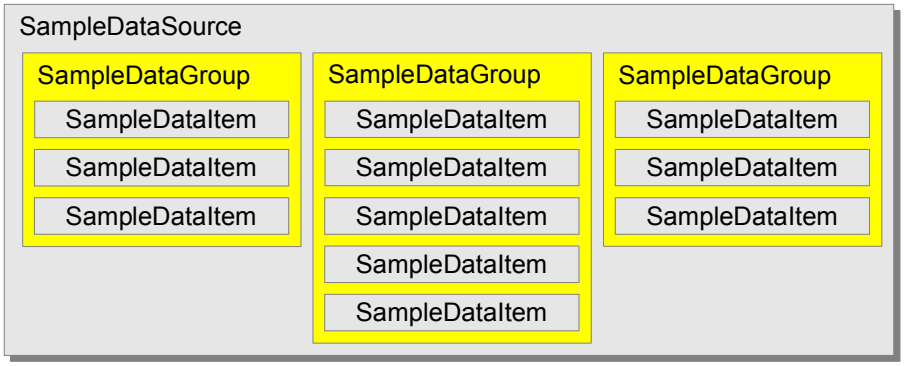
Wie Sie sehen, wird mittels *Frame*-Objekt zwischen beiden Seiten-Instanzen gewechselt, mehr dazu im Abschnitt 18.3 ab Seite 915.

18.2.2 Geteilte App (Split App)

Mit dem Template *Geteilte App* steht Ihnen ein etwas komplexerer Anwendungstyp zur Verfügung, der jedoch auch schon recht speziell ist. So können Sie damit Apps wie Newsreader, E-Mail-Apps etc. erstellen. Ausgehend von einer Übersichtsseite (*ItemsPage*) mit verschiedenen Gruppen steuern Sie den Zugriff auf mehrere *SplitPage*-Instanzen. In diesen wird links die Liste der Items und rechts der Inhalt des jeweilig gewählten Items dargestellt:



Die komplette Navigation zwischen den Seiten/Einträgen ist bereits implementiert, die Daten für die Darstellung werden aus einem vorgefertigten Datenmodell (Verzeichnis *DataModel*, Klassen *SampleDataSource*, *SampleDataGroup* und *SampleDataItem*) entnommen:



Damit wird auch klar, wie Ihre Vorgehensweise bei eigenen Anwendungen ist:

1. Anpassen der Klassen *SampleDataGroup* und *SampleDataItem* an Ihre Erfordernisse
2. Implementieren einer Routine zum Laden der Daten (siehe Konstruktor der Klasse *SampleDataSource*). Diese Daten können beispielsweise von einem Webdienst oder aus lokalen Daten stammen.
3. Anpassen der Symbole (Verzeichnis *Assets*, *DarkGray.png*, *MediumGray.png* ...). Auch hier können Sie dafür sorgen, dass diese Symbole erst geladen werden, wenn auf die betreffende Eigenschaft zugegriffen wird (Klasse *SampleDataCommon*, Eigenschaft *Image*). Gleiches trifft auf die Eigenschaft *Description* zu. So vermeiden Sie unnötigen Traffic beim Start der App.

Bei einem Blick hinter die Kulissen (*ItemsPage.xaml.cs*) werden Sie feststellen, dass hier keinerlei Instanz der Klasse *SampleDataSource* erzeugt und zugewiesen wird. Dies ist auch nicht nötig, die Instanziierung erfolgt im XAML-Code der Seite:

```
<common:LayoutAwarePage
...
    <UserControl.Resources>
        <CollectionViewSource
            x:Name="itemsViewSource"
            Source="{Binding Items}"
            d:Source="{Binding ItemGroups, Source={d:DesignInstance Type=data:SampleDataSource,
           IsDesignTimeCreatable=True}}"/>
    </UserControl.Resources>
...
             <GridView
                x:Name="itemGridView"
                AutomationProperties.AutomationId="ItemsGridView"
                AutomationProperties.Name="Items"
                Margin="116,0,116,46"
```

Zuweisen der Datenquelle:

```
                ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
```

Zuweisen des *ItemTemplate*:

```
                ItemTemplate="{StaticResource Standard250x250ItemTemplate}"
                SelectionMode="None"
                IsItemClickEnabled="True"
                ItemClick="ItemView_ItemClick"/>
...
```

Doch wie haben Sie Einfluss auf die Gestaltung und das Aussehen der Seite bzw. der Listendarstellung? In diesem Fall sollten Sie nicht nur die Datei *ItemsPage.xaml* (Grundlayout der Seite) betrachten, sondern auch einen Blick auf die Datei *StandardStyles.xaml* aus dem Verzeichnis *Common* werfen. Hier finden Sie die Definition der DataTemplates (diese definieren das Aussehen der Items innerhalb der Liste):

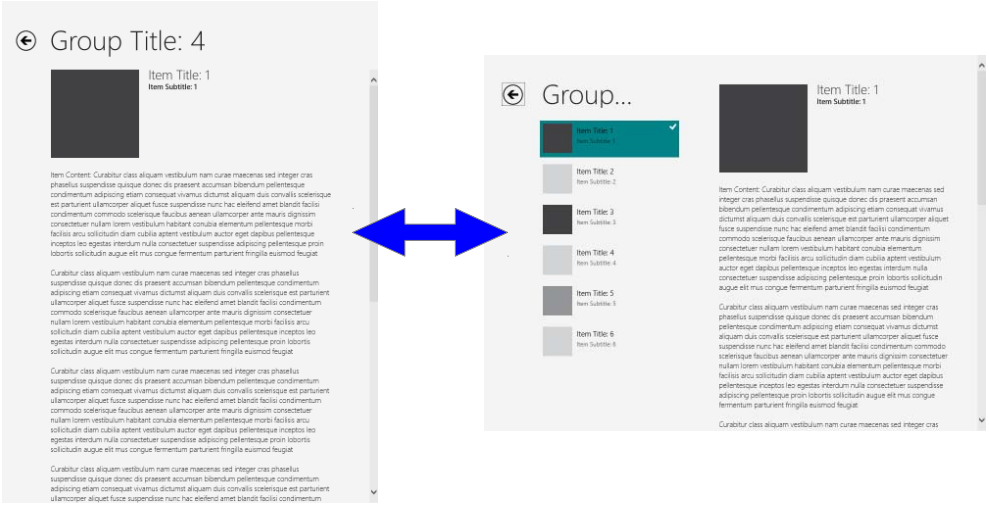
```
...
    <DataTemplate x:Key="Standard250x250ItemTemplate">
        <Grid HorizontalAlignment="Left" Width="250" Height="250">
            <Border Background="{StaticResource ListViewItemPlaceholderRectBrush}">
                <Image Source="{Binding Image}" Stretch="UniformToFill"/>
            </Border>
            <StackPanel VerticalAlignment="Bottom"
                        Background="{StaticResource ListViewItemOverlayBackgroundBrush}">
                <TextBlock Text="{Binding Title}"
                           Foreground="{StaticResource ListViewItemOverlayTextBrush}"
                           Style="{StaticResource TitleTextStyle}"
                           Height="60" Margin="15,0,15,0"/>
                <TextBlock Text="{Binding Subtitle}"
```

```
                        Foreground="{StaticResource ListViewItemOverlaySecondaryTextBrush}"
                        Style="{StaticResource CaptionTextStyle}"
                        TextWrapping="NoWrap" Margin="15,0,15,10"/>
            </StackPanel>
        </Grid>
    </DataTemplate>
...
```

Ändern Sie in dieser Datei die Gestaltung der einzelnen Listen-Einträge oder erstellen Sie eigene Templates. Doch Achtung:

HINWEIS: Denken Sie daran, dass Ihre App auch mit weniger Platz auskommen muss, wenn das Ausgabegerät beispielsweise gedreht wird.

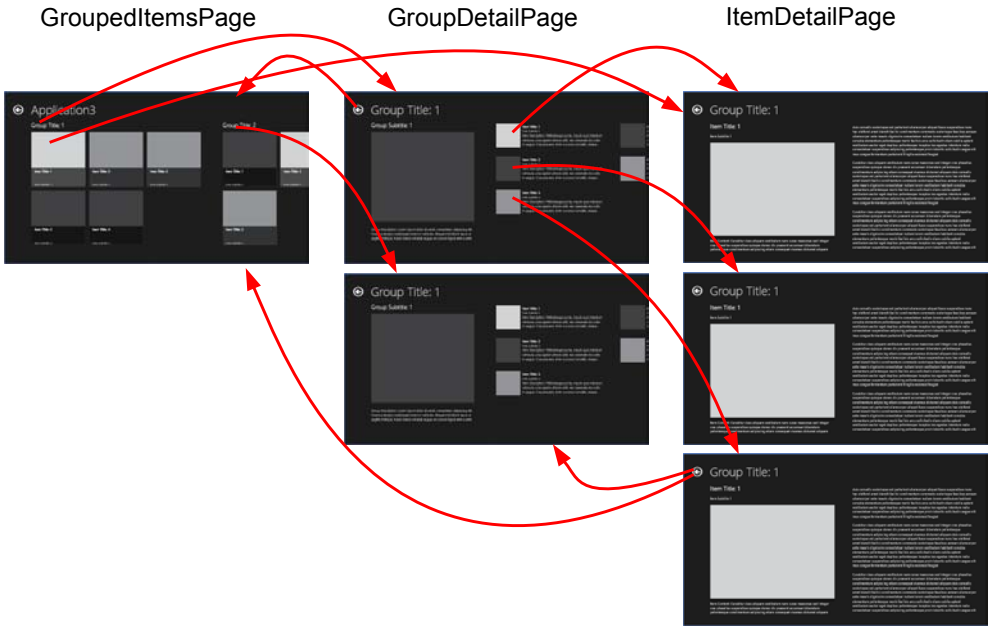
Das Projekt-Template hat bereits für diesen Fall vorgesorgt. So wird beispielsweise das *ItemTemplate* der *SplitPage* durch einen *VisualStateManager* ausgetauscht, wenn sich der Ansichtsmodus ändert. Der Nutzer sieht ein anderes Layout der Seite:



Mehr dazu später, an dieser Stelle nur so viel: Sie müssen auch diese alternativen Templates verändern, wenn Sie sich an der Datei *StandardStyles.xaml* austoben wollen.

18.2.3 Raster-App (Grid App)

Wem das Template *Geteilte App* noch nicht komplex genug ist, der sollte sich einmal die Rasteranwendung ansehen. Diese arbeitet zwar mit dem gleichen Datenmodell wie die *Geteilte Anwendung* (*SampleDataSource*, *SampleDataGroup* und *SampleDataItem*), stellt jedoch wesentlich erweiterte Navigationsmöglichkeiten zwischen Übersichtsseite (*GroupedItemsPage*), Gruppenseiten (*GroupDetailPage*) und den eigentlichen Detailseiten (*ItemDetailPage*) zur Verfügung.



Wie bereits erwähnt, auch hier müssen Sie sich nicht mehr um die Navigation kümmern, "lediglich" Layout und Content müssen noch hinzugefügt werden. Die Vorgehensweise hatten wir bereits im vorhergehenden Abschnitt 18.2.2 besprochen.

An dieser Stelle wollen wir uns nicht weiter mit diesem recht komplexen Template beschäftigen, da es doch schon ziemlich speziell ist. Wenden wir uns deshalb einigen Seiten-Templates zu.

18.2.4 Leere Seite (Blank Page)

Diesen Seitentyp, der direkt auf der Klasse *Page* basiert, hatten wir bereits mehrfach verwendet.

```
<Page
    x:Class="AppBarControl.BlankPage1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AppBarControl"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>
```

Ergebnis obigen XAML-Codes ist eine wirklich leere Seite mit schwarzem Hintergrund.

Zu mehr als einem Ausgangspunkt für eigene Seitenlayouts taugt dieses Template nicht, dies vor allem auch, weil dieser Seitentyp über keinen vordefinierten *VisualStateManager* verfügt, der auf die verschiedenen Ansichtsmodi (Bildauflösung des Ausgabegeräts, Drehen) reagiert.

18.2.5 Standardseite (Basic Page)

Dieses Seiten-Template besitzt bereits die "Lizenz zum Navigieren" und bietet rudimentäre Unterstützung für die Seitenanpassung beim Andocken der App.

Realisiert wird diese Funktionalität über die Klasse *LayoutAwarePage* (Code im Unterverzeichnis *Common*) und einem *VisualStateManager* im XAML-Code der Seite.

Der XAML-Code der Standardseite:

```
<common:LayoutAwarePage
    x:Name="pageRoot"
    x:Class="AppBarControl.BasicPage1"
...
    mc:Ignorable="d">

    <Page.Resources>
```

Hier können Sie die Überschrift der Seite festlegen, alternativ ist dies auch zentral über die Datei *App.xaml* möglich:

```
        <x:String x:Key="AppName">My Application</x:String>
    </Page.Resources>
```

Grundlayout der Seite ist ein *Grid* mit zwei Zeilen. Zeile 1 enthält den Back-Button und die Überschrift, Zeile 2 steht Ihnen zur freien Verfügung:

```
    <Grid Style="{StaticResource LayoutRootStyle}">
        <Grid.RowDefinitions>
            <RowDefinition Height="140"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
```

Der Seitenkopf mit dem Back-Button und der Seitenüberschrift:

```
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Button x:Name="backButton" Click="GoBack" IsEnabled="{Binding Frame.CanGoBack,
                    ElementName=pageRoot}" Style="{StaticResource BackButtonStyle}"/>
            <TextBlock x:Name="pageTitle" Grid.Column="1" Text="{StaticResource AppName}"
                    Style="{StaticResource PageHeaderTextStyle}"/>
        </Grid>
```

Der folgende *VisualStateManager* definiert Storyboards für die verschiedenen App-Darstellungen (Vollbild, Angedockt ...):

```
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="ApplicationViewStates">
```

Dies ist die Standarddarstellung der App (Querformat und voller Bildschirm):

```
                <VisualState x:Name="FullScreenLandscape"/>
```

Die App wird durch eine angedockte App in der Breite reduziert:

```
                <VisualState x:Name="Filled"/>
```

Hier finden sich Anpassungen (Style-Änderung für den Back-Button), wenn das Ausgabegerät gedreht wird:

```
                <VisualState x:Name="FullScreenPortrait">
                    <Storyboard>
                        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                            Storyboard.TargetProperty="Style">
                            <DiscreteObjectKeyFrame KeyTime="0"
                                Value="{StaticResource PortraitBackButtonStyle}"/>
                        </ObjectAnimationUsingKeyFrames>
                    </Storyboard>
                </VisualState>
```

Die App ist angedockt, d.h., die Breite wurde drastisch reduziert:

```
                <VisualState x:Name="Snapped">
                    <Storyboard>
```

Über das Storyboard werden jetzt der Back-Button und die Überschrift in der Größe angepasst:

```
                        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                            Storyboard.TargetProperty="Style">
                            <DiscreteObjectKeyFrame KeyTime="0"
                                 Value="{StaticResource SnappedBackButtonStyle}"/>
                        </ObjectAnimationUsingKeyFrames>
                        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="pageTitle"
                                                       Storyboard.TargetProperty="Style">
                            <DiscreteObjectKeyFrame KeyTime="0"
                                  Value="{StaticResource SnappedPageHeaderTextStyle}"/>
                        </ObjectAnimationUsingKeyFrames>
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
    </Grid>
</common:LayoutAwarePage>
```

Testen Sie ruhig mal eine Seite, die auf diesem Template basiert. Verfügt das Projekt nur über eine derartige Seite bzw. handelt es sich um die Startseite, wird kein Back-Button angezeigt. Beim Wechsel zwischen zwei Seiten wird automatisch der Back-Button aktiviert. Nutzen Sie auch einmal den Simulator und drehen Sie die Ansicht bzw. docken Sie die App an. Sie werden feststellen, dass Überschrift und Back-Button in diesem Fall kleiner werden.

18.2.6 Ein eigenes Grundlayout erstellen

Angelehnt an die Microsoft-Beispiele und das Template *Standardseite* wollen wir ein etwas erweitertes Layout erstellen, das zum einen über eine Fußzeile, zum anderen aber auch über einen etwas flexibleren Clientbereich verfügt, der in der angedockten Ansicht nicht gleich umbricht:

```
<common:LayoutAwarePage
...
 <Grid Style="{StaticResource LayoutRootStyle}">
```

Wir erweitern das zentrale *Grid* um eine weitere Zeile. Beachten Sie, dass die Höhe sowohl der oberen, als auch der unteren Zeile vom enthaltenen Content bestimmt sind:

```
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
```

Für den Seitenkopf nutzen wir weiterhin ein *StackPanel*:

```
    <StackPanel x:Name="Header" Grid.Row="0">
...
    </StackPanel>
```

Den eigentlichen Inhalt der Seite verlegen wir in einen *ScrollViewer*:

```
    <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled">
...
    </ScrollViewer>
```

Der Vorteil dieser Variante: auch in der gedrehten Ansicht steht den enthaltenen Controls immer genügend Platz zur Verfügung, im Zweifel muss der Anwender eben die Scrollbars verwenden.

Die Fußzeile wird ebenfalls als *StackPanel* implementiert:

```
    <StackPanel x:Name="Footer" Grid.Row="2" Margin="0, 10, 0, 0" VerticalAlignment="Bottom">
...
    </StackPanel>
  </Grid>
...
```

Möchten Sie dieses Grundlayout mehrfach verwenden, können Sie dieses auch als Template abspeichern (*Datei|Vorlage exportieren|Elementvorlage*). Im Weiteren stehen Ihnen dann das Formular bzw. die Page über *Projekt|Neues Element hinzufügen* zur Verfügung.

HINWEIS: Die weiteren Buchbeispiele basieren weitgehend auf diesem Layout, zur Vereinfachung haben wir unserem Layout noch eine Beenden-Taste spendiert, so müssen Sie nicht erst warten bis Windows die Anwendung aus dem Speicher wirft.

18.3 Seitenauswahl und -navigation

Neben dem Entwurf von Seiten stellt sich früher oder später auch die Frage nach der Navigation zwischen diesen Seiten. Im Gegensatz zu den Windows Forms und den WPF-Anwendungen müssen Sie hier grundsätzlich umdenken: aktiv und sichtbar ist immer nur eine Seite, wie schon im Abschnitt 18.1.2 beschrieben.

18.3.1 Die Startseite festlegen

In den bisherigen Beispielen hatte wir es schon mehrfach erwähnt, der Eintrittspunkt für Ihre WinRT-App liegt in der Datei *App.xaml.cs*, genauer in der Methode *OnLaunched*.

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame == null)
    {
        rootFrame = new Frame();
        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        { }
        Window.Current.Content = rootFrame;
    }
```

An dieser Stelle legen Sie die Startseite fest:

```
    if (rootFrame.Content == null)
    {
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
        {
            throw new Exception("Failed to create initial page");
        }
    }
    Window.Current.Activate();
}
```

HINWEIS: Beachten Sie, dass Sie hier lediglich den Typ der Seite angeben müssen, es wird keine Instanz übergeben, diese wird intern durch die Methode *Navigate* erzeugt.

18.3.2 Navigation und Parameterübergabe

Verfügt Ihre App über mehr als eine Seite, was meist der Fall sein wird, so stehen Sie zum einen vor der Aufgabe, einen Seitenwechsel zu realisieren, zum anderen müssen häufig auch Informationen an die aufgerufene Seite übergeben werden.

Die eigentliche Navigation realisieren Sie über den der aktuellen Seite übergeordneten *Frame*, indem Sie dessen *Navigate*-Methode aufrufen.

BEISPIEL 18.3: Seitenwechsel als Reaktion auf das *Click*-Ereignis

C#
```
private void HyperlinkButton_Click_1(object sender, RoutedEventArgs e)
{
    this.Frame.Navigate(typeof(BasicPage2));
}
```

Eine Überladung der *Navigate*-Methode ermöglicht zusätzlich die Übergabe von Parametern, dies können einfache Eigenschaften oder auch gleich die aktuelle Page-Instanz sein.

BEISPIEL 18.4: Parameterübergabe

C#
```
this.Frame.Navigate(typeof(BasicPage2), textBox1.Text);
```

oder auch

```
this.Frame.Navigate(typeof(BasicPage2), this);
```

Die Auswertung erfolgt in der *OnNavigatedTo*-Methode der aufgerufenen Page:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    string inhaltTextBox = e.Parameter as string;
```

oder auch

```
    BasicPage1 page1 = e.Parameter as BasicPage1;
```

HINWEIS: Vergessen Sie nicht, dass Sie für den externen Zugriff auf die Seitenelemente zunächst auch den *FieldModifier* anpassen müssen.

BEISPIEL 18.5: Änderung *FieldModifier* für eine *TextBox*

C#
```
...
  <TextBox Text="Hallo" Name="TextBox1" x:FieldModifier="public"></TextBox>
...
```

Ein komplettes Beispiel finden Sie im Praxisbeispiel

▶ 18.6.1 Seitennavigation und Parameterübergabe

18.3.3 Den Seitenstatus erhalten

Probieren Sie obiges Beispiel aus, können Sie problemlos zwischen den Seiten wechseln, allerdings dürfte Ihnen schnell auffallen, dass die Eingaben auf der Ausgangsseite nach der Rückkehr von einer anderen Seite wieder gelöscht sind. Dies kann gewollt sein, in den meisten Fällen dürfte aber der Nutzer von diesem Verhalten wenig erfreut sein, da dies bei Desktop-Anwendungen eher selten vorkommt.

Der Grund für dieses Verhalten liegt in der Funktionsweise des übergeordneten Frames. Bei jedem Aufruf der *Navigate*-Methode wird standardmäßig eine neue Instanz der Seite erstellt. Alternativ

können Sie die *NavigationCacheMode*-Eigenschaft nutzen, um die Instanz im Speicher zu halten. Setzen Sie den Wert für die betreffende Seite einfach auf *Enabled*:

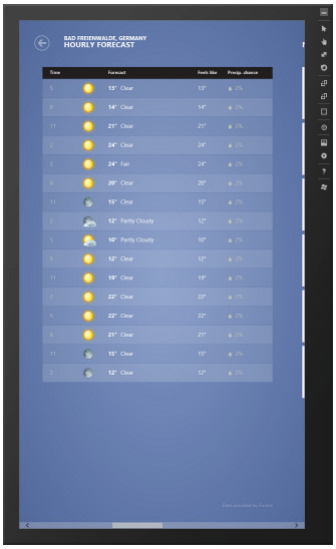
```
<common:LayoutAwarePage
    x:Name="pageRoot"
    x:Class="NavigationParameter.BasicPage1"
...
    mc:Ignorable="d" NavigationCacheMode="Enabled" >
...
```

18.4 Die vier App-Ansichten

Mehrfach wurde bereits erwähnt, dass WinRT-Apps immer exklusiv und im Vollbild dargestellt werden. Dies ist prinzipiell richtig, aber ...

18.4.1 Vollbild quer und hochkant

Die beiden folgenden Abbildungen zeigen die zwei Normal-Zustände der Vollbild-App (Sie erinnern sich: das Ausgabegerät kann durch den Nutzer auch spontan gedreht werden):



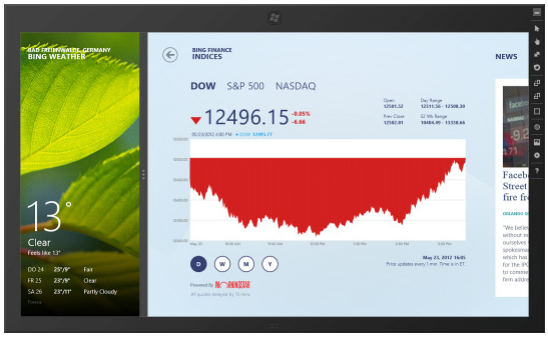
Es dürfte schnell ersichtlich sein, dass Sie in einigen Fällen auf diesen Formatwechsel reagieren müssen, halbe Überschriften und nicht komplett angezeigte Grafiken/Controls sehen einfach nicht sehr toll aus.

In jedem Fall verfügen WinRT-Apps über eine Mindestauflösung von 1024x768 bzw. 768x1024 Pixeln. Sie müssen sich also nicht mehr den Kopf darüber zerbrechen, wie Ihre App bei einer Bildschirmauflösung von nur 640x480 Pixel aussieht.

Wie Sie auf den Formatwechsel reagieren können, zeigen wir Ihnen im Abschnitt 18.4.3, zunächst sehen wir uns noch die beiden anderen Zustände der App an.

18.4.2 Angedockt und Füllmodus

Die folgende Abbildung zeigt gleich zwei weitere App-Zustände, links sehen Sie eine "angedockte App" (Snapped), rechts eine App im Füllmodus (Filled):



HINWEIS: Das Andocken in der WinRT-Oberfläche ist erst ab einer Auflösung von 1366x768 Pixeln möglich! In diesem Fall verfügt die angedockte App über eine Mindestbreite von 320 Pixeln (bei höheren Bildschirmauflösungen entsprechend mehr).

Arbeitet Ihre App im Füllmodus (quasi der Lückenbüßer), muss Sie mit dem noch verfügbaren Platz klarkommen. In den meisten Fällen sollte dies kaum ein Problem sein, hier kann durch geschicktes Auswählen von Seitenlayouts (*StackPanel* o.ä.) problemlos auf die Größenänderung reagiert werden.

Ganz anders bei der angedockten App. Hier ist dann schon eine ganze Menge Arbeit angesagt, denn mit dem einfachen Verschieben von Controls werden Sie kaum das gewünschte Resultat erreichen. Ziel muss es sein, die Informationen Ihrer App in einem schmalen Hochkantformat zu bündeln und auf überflüssige Informationen zu verzichten. Ob dies immer funktioniert sei dahingestellt, ein Spiel im angedockten Modus dürfte in vielen Fällen sinnfrei sein, hier wäre es besser, einen Wechsel in diesen Modus zu verhindern (siehe 18.4.4).

18.4.3 Reagieren auf die Änderung

Problem erkannt, Gefahr gebannt! Doch wie soll das eigentlich vonstatten gehen? Hier hilft Ihnen zum einen das *SizeChanged*-Ereignis der aktuellen Seite weiter, zum anderen ein *VisualStateManager*. Ersteres dient der Erkennung einer Größenänderung (egal ob Drehung, Zugriff per Remote-Desktop, Andocken etc.), letzteres bietet eine bequeme Lösung für die Reaktion auf diese Größenänderung.

Haben Sie eine auf dem Standardseiten-Template basierende Page (Klasse *LayoutAwarePage*) erstellt, ist die nötige "Infrastruktur" bereits implementiert, d h. es genügt wenn Sie Anpassungen daran vornehmen. Dazu nutzen Sie den im nachfolgenden beschrieben *VisualStateManager*.

SizeChanged auswerten

Verwenden Sie eine leere Seite, fügen Sie dieser einfach einen Eventhandler für das *SizeChanged*-Ereignis hinzu. Ursache für das Auftreten des Ereignisses können Änderungen der Bildschirmauflösung, Andock-Manöver, Geräterotationen, Zugriff per Remote-Desktop etc. sein. Für Sie wichtig:

HINWEIS: Über den Parameter *e.NewSize* erhalten Sie Auskunft über die neue Darstellungsgröße, über *e.PreviousSize* können Sie die vorhergehende Größe abfragen.

Bevor Sie jetzt mühevoll anfangen, aus diesen Werten die Drehung oder ähnliches "herauszulesen" – vergessen Sie es gleich wieder. Dabei hilft Ihnen wesentlich effizienter *Windows.UI.ViewManagement.ApplicationView.Value*.

Wert	Beschreibung
FullScreenLandscape	Vollbilddarstellung im Querformat.
Filled	Füllmodus, die App nutzt den Platz den ihr eine angedockte Anwendung lässt.
Snapped	Angedockt, die App selbst ist am linken oder rechten Bildrand angedockt.
FullScreenPortrait	Vollbilddarstellung im Hochkantformat.

BEISPIEL 18.6: Direktes Auswerten des Andockens

C#

```
private void Page_SizeChanged_1(object sender, SizeChangedEventArgs e)
{
    Windows.UI.ViewManagement.ApplicationViewState vs =
                Windows.UI.ViewManagement.ApplicationView.Value;
    switch (vs)
    {
        case Windows.UI.ViewManagement.ApplicationViewState.Snapped:
            frm1.Navigate(typeof(Angedockt));
            break;
        case Windows.UI.ViewManagement.ApplicationViewState.Filled:
        case Windows.UI.ViewManagement.ApplicationViewState.FullScreenLandscape:
        case Windows.UI.ViewManagement.ApplicationViewState.FullScreenPortrait:
        default:
            frm1.Navigate(typeof(Normalansicht));
            break;
    }
}
```

In diesem Fall wird, basierend auf dem geänderten Ansichtsstatus, der App ein anderer Frameinhalt auf der aktuellen Seite angezeigt.

Ein komplettes Beispiel zum Thema finden Sie unter

► 18.6.2 Auf Ansichtsänderungen reagieren

VisualStateManager

Nicht immer wollen oder können Sie gleich den kompletten Inhalt Ihrer Seite ändern, nur weil die App angedockt wird. Teilweise genügt es auch, wenn Controls an die neuen Gegebenheiten angepasst werden. Dazu bietet sich ein sogenannten *VisualStateManager* an.

Der *VisualStateManager* verwaltet Zustandsänderungen. Eine solche kann benutzt werden, um verschiedene Arten von Effekten zu realisieren. Dies kann ein zustandsbasierter Effekt sein oder auch eine Transition. Im vorliegenden Fall soll beim Eintritt eines Ereignisses ein Storyboard ausgeführt werden (z.B. Schriftänderungen, Margin-Änderungen etc.). Der Vorteil dieser Vorgehensweise: Sie können im XAML-Code die neuen Zustände bzw. die dazu nötigen Storyboards definieren, den Zustandswechsel lösen Sie per C#-Code aus.

Arbeiten Sie mit dem Template *Leere Seite*, müssen Sie sich um das Erstellen und die Ansteuerung des *VisualStateManager* selbst kümmern. Seiten, die auf dem Template *Standardseite* basieren, verfügen bereits über die erforderliche Infrastruktur.

BEISPIEL 18.7: Der *VisualStateManager* in den Standardseiten

In der Klassendefinition von *LayoutAwarePage* findet sich die Methode *InvalidateVisualState*, die unter anderem von den Ereignissen *ViewStateChanged* und *SizeChanged* aufgerufen wird:

```
public void InvalidateVisualState(ApplicationViewState? viewState = null)
{
    if (this._layoutAwareControls != null)
    {
        string visualState = DetermineVisualState(viewState == null ?
              ApplicationView.Value : viewState.Value);
        foreach (var layoutAwareControl in this._layoutAwareControls)
        {
```

Hier wird der Zustandswechsel bei der Ansicht an den *VisualStateManager* weitergereicht:

```
            VisualStateManager.GoToState(layoutAwareControl, visualState, false);
        }
    }
}
```

Die Definition des *VisualStateManager* finden Sie im XAML-Code der Seite:

```
...
    <VisualStateManager.VisualStateGroups>
```

Die verschiedenen Zustände korrespondieren mit den bereits bekannten *ApplicationView*-Werten:

```
        <VisualStateGroup>
            <VisualState x:Name="FullScreenLandscape"/>
```

BEISPIEL 18.7: Der *VisualStateManager* in den Standardseiten

```
                <VisualState x:Name="Filled"/>
                <VisualState x:Name="FullScreenPortrait">
```

Die Ansicht ändert sich in eine Hochkantdarstellung, das folgende Storyboard wird ausgeführt:

```
                    <Storyboard>
```

Für den Back-Button wird die Eigenschaft *Style* geändert:

```
                        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                            Storyboard.TargetProperty="Style">
                            <DiscreteObjectKeyFrame KeyTime="0"
                               Value="{StaticResource PortraitBackButtonStyle}"/>
                        </ObjectAnimationUsingKeyFrames>
                    </Storyboard>
                </VisualState>
```

HINWEIS: Durch Änderung des Styles können Sie gleichzeitig auf viele einzelne Eigenschaften und damit auf das komplette Aussehen eines Controls Einfluss nehmen. Sie müssen also, wie in obigem Fall, nur Styles für die einzelnen Controls definieren und beim Ansichtswechsel neu zuweisen.

Beim Andocken werden wiederum andere Styles zugewiesen:

```
                <VisualState x:Name="Snapped">
                    <Storyboard>
                        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                              Storyboard.TargetProperty="Style">
                            <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
                               SnappedBackButtonStyle}"/>
                        </ObjectAnimationUsingKeyFrames>
...
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
```

Beachten Sie, dass Sie den Ursprungszustand nicht erneut definieren müssen, dieser ist ja bereits beim Entwurf im XAML-Code abgelegt worden.

Ein komplettes Beispiel zum Thema finden Sie unter

- 18.6.2 Auf Ansichtsänderungen reagieren

18.4.4 Angedockten Modus aktiv beenden

In einigen Fällen ist es erforderlich, die App aktiv in den Vollbildmodus zu bringen, wenn diese vorher angedockt war (z.B. zum Öffnen von Dateidialogen). Aus diesem Grund bietet die statische *ApplicationView*-Klasse die Methode *TryUnsnap* an, mit der Sie Ihre App wieder in den Vollbildmodus bringen können.

BEISPIEL 18.8: Verwendung von *TryUnsnap*

C#
```
using Windows.UI.ViewManagement;
...
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            ApplicationView.TryUnsnap();
...
        }
```

HINWEIS: Im *SizeChanged*-Ereignis hat der Aufruf der Methode normalerweise keine Auswirkung, es sei denn, Sie rufen die Methode verzögert auf.

BEISPIEL 18.9: Docking per Task beenden

C#
```
        private void pageRoot_SizeChanged_1(object sender, SizeChangedEventArgs e)
        {
            Task.Factory.StartNew(() =>
                {
                   Task.Delay(1000).Wait();
                   if (ApplicationView.Value == ApplicationViewState.Snapped)
                       ApplicationView.TryUnsnap();
                });
        }
```

Auf die kleine Zeitverzögerung können wir leider nicht verzichten, die App dockt also zunächst noch kurz an, springt dann aber wieder in den Vollbildmodus zurück.

18.5 Skalieren von Apps

Nicht genug der Neuerungen – WinRT bietet auch umfassende Unterstützung für hochauflösende Displays. Wer bereits damit gearbeitet hat, wird festgestellt haben, dass bei kleinen Bildschirmdiagonalen mit zunehmender Auflösung die Lesbarkeit von Schriften schnell leidet. Diese müssen dann entsprechend vergrößert werden. Da gerade Tablets mit begrenzten Diagonalen aufwarten und teilweise mit hohen Auflösungen arbeiten, bietet Windows 8 eine Skalierung in drei Prozentstufen:

- 100%
- 140% auf HD-Tablets
- 180% auf Quad-XGA-Tablets

Doch wie stellen Sie eigentlich fest, mit welcher Auflösung gearbeitet wird? Nichts leichter als dies, alle relevanten Informationen über die Anzeige finden Sie in der statischen Klasse *DisplayProperties*.

BEISPIEL 18.10: Auswerten von Skalierungsänderungen und Anzeige der aktuellen Skalierung in einem *TextBlock*.

C#
```
...
        public BasicPage1()
        {
            this.InitializeComponent();
            DisplayProperties.LogicalDpiChanged +=DisplayProperties_LogicalDpiChanged;
            tb1.Text = "ResolutionScale: " + DisplayProperties.ResolutionScale.ToString();
            tb2.Text = "LogicalDpi: " + DisplayProperties.LogicalDpi.ToString();
        }

        void DisplayProperties_LogicalDpiChanged(object sender)
        {
            tb1.Text = "ResolutionScale: " + DisplayProperties.ResolutionScale.ToString();
            tb2.Text = "LogicalDpi: " + DisplayProperties.LogicalDpi.ToString();
        }
...
```

Während *ResolutionScale* die Werte 100, 140 oder 180 zurückgibt, liefert *LogicalDpi* die Auflösung des Displays (bei einem Desktopmonitor 96).

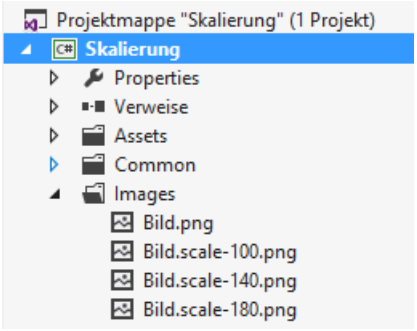
HINWEIS: *ResolutionScale* entspricht **nicht** den Einstellungen in der Systemsteuerung, diese wirken sich nur auf den Desktop und nicht auf WinRT-Apps aus.

Doch was ist eigentlich mit Grafiken? Wie die Schriftarten, werden auch diese vom System entsprechend der obigen Einstellung skaliert. Solange es sich um Vektorgrafiken handelt (XAML, SVG) ist das kein Problem. Doch wer schon einmal Icons für Schaltflächen verwendet und per Stretch auf eine bestimmt Größe skaliert hat, wird festgestellt haben, dass die Qualität zu wünschen übrig lässt (unscharfe Ränder).

Für Ihre WinRT-App gibt es aus diesem Grund eine recht intuitive Möglichkeit, Ressourcen für die drei oben genannten Skalierungen in der Anwendung zu hinterlegen. Speichern Sie einfach mehrere Versionen mit unterschiedlicher Auflösung, die der folgenden Namenskonvention folgen:

```
<Bildname>.scale-100.<Extension>
<Bildname>.scale-140.<Extension>
<Bildname>.scale-180.<Extension>
```

Siehe auch folgende Abbildung:



Bei der Bildauswahl (*Source*-Eigenschaft) geben Sie im obigen Fall nur "Bild.png" an. Beim Start der App wird später automatisch das Bild gewählt, das der aktuellen Skalierung am nächsten kommt.

Testen können Sie dieses Verhalten, indem Sie in den WinRT-PC-Einstellungen unter "Erleichterte Bedingung" die Option "Alle Elemente auf dem Desktop vergrößern" aktivieren. In diesem Fall wird das Bild mit der Benennung ".scale-140"ausgewählt und angezeigt.

HINWEIS: Neben der automatischen Wahl von Grafiken in Abhängigkeit von der Skalierung gibt es auch die Möglichkeit, je nach Landeseinstellung andere Bilder anzuzeigen oder die Kontrasteinstellung des Systems auszuwerten.

18.6 Praxisbeispiele

18.6.1 Seitennavigation und Parameterübergabe

Die folgende kleine App besteht aus zwei Seiten: *BasicPage1* und *BasicPage2*. Ziel ist die Übergabe von Werten (z.B. Inhalt einer *TextBox*) von *BasicPage1* an *BasicPage2*.

Oberfläche BasicPage1

HINWEIS: Wir verwenden unser Standardlayout aus dem Abschnitt 18.2.6.

```
...
       <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                 Grid.Row="1" ZoomMode="Disabled">
          <StackPanel Margin="120,40,0,0">
```

Kurze Beschriftung:

```
              <TextBlock Style="{StaticResource BodyTextStyle}">
```

```
                Bitte geben Sie hier einen Text ein:
            </TextBlock>
```

Eine *TextBox* für die zu übergebenden Daten (beachten Sie die Verwendung von *FieldModifier*[1]):

```
            <TextBox Name="textBox1" Margin="0,10,12,50" x:FieldModifier="public"/>
```

HINWEIS: Sie können natürlich auch eine komplexere Seite erstellen und diese in *BasicPage2* auswerten.

Mit dem Button wechseln wir die Seite:

```
            <Button Width="302" Content="Gehe zur Seite 2 ..."
                    Style="{StaticResource TextButtonStyle}"  Click="Button_Click_2"/>
        </StackPanel>
    </ScrollViewer>
...
```

Quellcode BasicPage1

Mit dem Klick auf den Button rufen wir die *Navigate*-Methode auf und übergeben die komplette Instanz von *BasicPage1* als Parameter:

```
    private void Button_Click_2(object sender, RoutedEventArgs e)
    {
        this.Frame.Navigate(typeof(BasicPage2), this);
    }
```

HINWEIS: Wir hätten in diesem Fall auch nur die *Text*-Eigenschaft als Parameter übergeben können, aber in den meisten Fällen sind mehrere Werte interessant.

Oberfläche BasicPage2

HINWEIS: Wir verwenden unser Standardlayout aus dem Abschnitt 18.2.6.

```
...
    <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled">
            <StackPanel Margin="120,40,0,0">
```

Kurze Beschriftung:

```
            <TextBlock Style="{StaticResource BodyTextStyle}">
                Der Textbox-Inhalt auf Seite 1:</TextBlock>
```

Anzeige des übergebenen Wertes:

```
            <TextBlock Name="textBlock1" Style="{StaticResource SubheaderTextStyle}"/>
```

[1] Ohne diese Erweiterung kann von anderen Seiten nicht auf die *TextBox* zugegriffen werden.

```
            </StackPanel>
        </ScrollViewer>
...
```

Quellcode BasicPage2

Auswertung der Parameter in der *OnNavigatedTo*-Methode:

```
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            BasicPage1 page1 = e.Parameter as BasicPage1;
            textBlock1.Text = page1.textBox1.Text;
            base.OnNavigatedTo(e);
        }
```

Test

Nach dem Start der App können Sie auf der ersten Seite etwas Text eingeben, der nach dem Wechsel auf die zweite Seite dort ebenfalls angezeigt werden sollte.

Bemerkung

Wechseln Sie auf die erste Seite zurück, ist die *TextBox* wieder leer, da eine neue Instanz erstellt worden ist. Möchten Sie dieses Verhalten ändern, so setzen Sie die *NavigationCacheMode*-Eigenschaft der Seite auf *Enabled*:

```
<common:LayoutAwarePage
    x:Name="pageRoot"
    x:Class="NavigationParameter.BasicPage1"
...
    mc:Ignorable="d" NavigationCacheMode="Enabled" >
```

Jetzt bleiben die Inhalte der Seite erhalten, Sie können beliebig zwischen den Seiten umschalten.

18.6.2 Auf Ansichtsänderungen reagieren

Im diesem Praxisbeispiel wollen wir Ihnen zeigen, wie Sie auf das Ändern der App-Ansicht per C#- oder per XAML-Code reagieren können. Die App verfügt dazu über zwei zusätzliche Seiten *BasicPage2* und *BlankPage1*, die auf den entsprechenden Templates basieren.

Oberfläche & Quellcode MainPage

Fügen Sie der Oberfläche zwei Schaltflächen für den Start der weiteren Seiten hinzu:

```
...
        <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled">
            <StackPanel>
                <Button Grid.Column="1" Content="Reagieren per Code"  Margin="0,0,0,50"
                     Style="{StaticResource TextButtonStyle}" Click="Button_Click_2" />
```

```
            <Button Grid.Column="1" Content="Reagieren per VisualStateManager"
                Style="{StaticResource TextButtonStyle}" Click="Button_Click_3" />
        </StackPanel>
    </ScrollViewer>
```

Der dazugehörige Ereigniscode:

```
    private void Button_Click_2(object sender, RoutedEventArgs e)
    {
        this.Frame.Navigate(typeof(BlankPage1));
    }
    private void Button_Click_3(object sender, RoutedEventArgs e)
    {
        this.Frame.Navigate(typeof(BasicPage2));
    }
```

Oberfläche & Quellcode BlankPage1

Diese Seite verfügt über keinerlei *VisualStateManager*, Ansichtsänderungen werden nur per Code umgesetzt.

Zunächst die Oberfläche:

```
<ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
              Grid.Row="1" ZoomMode="Disabled" >
    <StackPanel>
```

Wesentlichstes Element ist ein *Frame*-Control, in den wir zur Laufzeit, je nach Ansichtsmodus, unterschiedliche Seiten einblenden werden:

```
            <Frame x:Name="frm1" Margin="0,50,0,0" MinHeight="200"/>
            <Button Grid.Column="1" Content="Zurück"  Margin="0,20,0,50"
                Style="{StaticResource TextButtonStyle}" Click="Button_Click_1" />
    </StackPanel>
</ScrollViewer>
```

Der Ereigniscode:

Nach dem Öffnen der Seite *BlankPage1* blenden wir zunächst im *Frame* die Seite *Normalansicht* ein[1]:

```
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        frm1.Navigate(typeof(Normalansicht));
    }
```

Auf einen möglichen Ansichtswechsel reagieren wir mit diesem Ereignishandler:

```
    private void Page_SizeChanged_1(object sender, SizeChangedEventArgs e)
    {
        Windows.UI.ViewManagement.ApplicationViewState vs =
```

[1] Hier sehen Sie auch recht schön das von den Webseiten bekannte Konzept der Frameprogrammierung.

```
                    Windows.UI.ViewManagement.ApplicationView.Value;
            switch (vs)
            {
                case Windows.UI.ViewManagement.ApplicationViewState.Snapped:
                    frm1.Navigate(typeof(Angedockt));
                    break;
                case Windows.UI.ViewManagement.ApplicationViewState.Filled:
                case Windows.UI.ViewManagement.ApplicationViewState.FullScreenLandscape:
                case Windows.UI.ViewManagement.ApplicationViewState.FullScreenPortrait:
                default:
                    frm1.Navigate(typeof(Normalansicht));
                    break;
            }
        }
```

Auch hier wechseln wir, je nach Ansicht, lediglich den Inhalts des Frames aus.

Für den Rücksprung auf die Startseite genügt der folgende Aufruf:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            this.Frame.GoBack();
        }
```

Oberfläche der beiden Frame-Seiten

Die Seite *Angedockt*:

```
...
    <Grid Background="LightBlue">
        <TextBlock Style="{StaticResource SubheaderTextStyle}" >Angedockt</TextBlock>
    </Grid>
...
```

Die Seite *Normalansicht*:

```
...
    <Grid Background="Yellow">
        <TextBlock Style="{StaticResource HeaderTextStyle}" >Normalansicht</TextBlock>
    </Grid>
...
```

HINWEIS: Wir beschränken uns auf unterschiedliche Hintergrundfarben und Schriftarten. Selbstverständlich können Sie hier auch zwei grundsätzlich unterschiedliche Seiten erstellen, die je nach Ansichtsmodus eingeblendet werden.

Oberfläche & Quellcode BasicPage2

Etwas einfacher haben wir es bei Verwendung eines VisualStateManagers.

Der Seiteninhalt:

```
...
  <ScrollViewer Name="scrollViewer1" VerticalScrollBarVisibility="Auto"
         HorizontalScrollBarVisibility="Auto" Grid.Row="1" ZoomMode="Disabled"
         Background="Yellow" Margin="120,50,40,50">
         <TextBlock Style="{StaticResource BodyTextStyle}">Mein Content ...</TextBlock>
  </ScrollViewer>
```

Der vordefinierte *VisualStateManager* (dieser wird intern durch das Ereignis *SizeChanged* angesteuert):

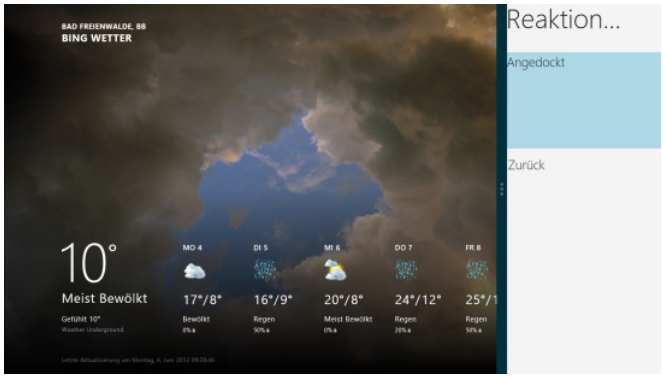
```
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup>
                    <VisualState x:Name="FullScreenLandscape"/>
                    <VisualState x:Name="Filled"/>
                    <VisualState x:Name="FullScreenPortrait">
                        <Storyboard>
                            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="backButton"
                                Storyboard.TargetProperty="Style">
                                <DiscreteObjectKeyFrame KeyTime="0" Value="{StaticResource
                                    PortraitBackButtonStyle}"/>
                            </ObjectAnimationUsingKeyFrames>
                        </Storyboard>
                    </VisualState>
                    <VisualState x:Name="Snapped">
                        <Storyboard>
...
```

Wir beeinflussen zusätzlich die Ränder des ScrollViewers:

```
                            <ObjectAnimationUsingKeyFrames Storyboard.TargetName="scrollViewer1"
                                Storyboard.TargetProperty="Margin">
                                <DiscreteObjectKeyFrame KeyTime="0">
                                    <DiscreteObjectKeyFrame.Value>
                                        <Thickness>40,20,40,20</Thickness>
                                    </DiscreteObjectKeyFrame.Value>
                                </DiscreteObjectKeyFrame>
                            </ObjectAnimationUsingKeyFrames>
                        </Storyboard>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
...
```

Test

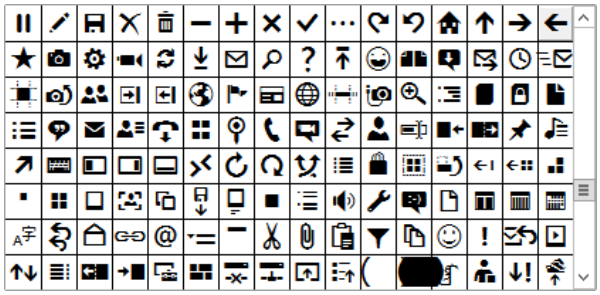
Starten Sie die App und testen Sie das Verhalten der beiden Seiten auf Ansichtswechsel durch das Andocken der App (siehe folgende Abbildung).



18.7 Tipps & Tricks

18.7.1 Symbole für WinRT-Oberflächen finden

Sollten Sie in die Verlegenheit kommen, Symbole für Ihre WinRT-Apps zu suchen, so werfen Sie zunächst einen Blick auf die schon vorhandenen Ressourcen von Windows 8. Verwenden Sie den Zeichensatz "Segoe UI Symbol", steht Ihnen eine ganze Reihe von häufig benötigten Symbolen zur Verfügung, die sich auch gleich in die "Designsprache" von WinRT einfügen:



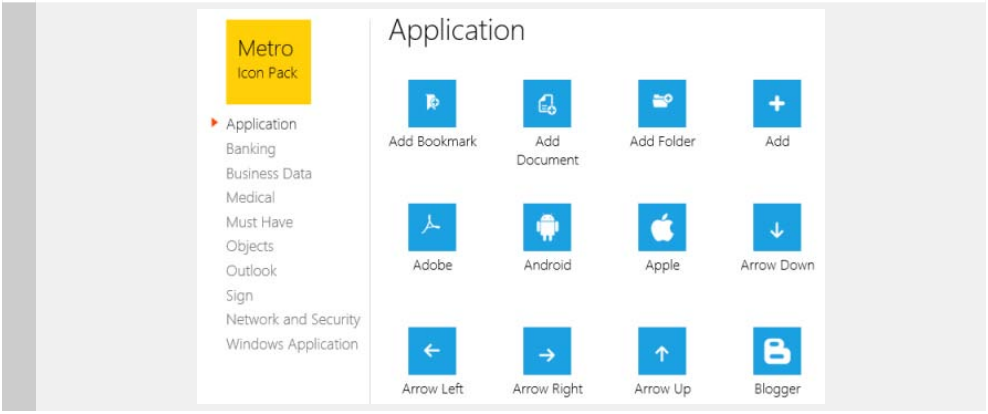
BEISPIEL 18.11: Verwendung im XAML-Code

XAML
```
<TextBlock Text="&#57615;" FontFamily="Segoe UI Symbol"/>
```

Frei verwendbare Alternativen für Symbole bieten die folgenden beiden Webseiten:

LINK: http://thenounproject.com/de/
LINK: http://www.syncfusion.com/downloads/metrostudio

BEISPIEL 18.12: Symbole von Syncfusion



18.7.2 Wie werde ich das Grufti-Layout schnell los?

Nicht jedem gefällt die grauenhaft dunkle Oberfläche, die Sie zunächst bei jedem neuen WinRT-Projekt als Standard angeboten bekommen. Bevor Sie jetzt anfangen, das Formular und die enthaltenen Controls einzeln anzupassen, sollten Sie einen Blick auf die Datei *App.xaml* werfen. Fügen Sie einfach den folgenden Eintrag hinzu, um allen Seiten Ihres Projekts ein etwas "freundlicheres" Äußeres zu geben:

```
<Application
    x:Class="Controls.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:MetroControls" RequestedTheme="Light" >
...
</Application>
```

Ohne großen Aufwand ändert sich so die komplette Farbpalette der Anwendung:

Kapitel 19

Die wichtigsten Controls

Nach einem ersten Rundflug über das Erstellen von WinRT-Apps wollen wir nun endlich zu den einzelnen Oberflächen-Controls kommen, die Sie zwar in der einen oder anderen Form bereits von WPF oder den Windows Forms her kennen, die jedoch teilweise ein etwas anderes Verhalten oder Aussehen aufweisen, was nicht zuletzt auch den Touch-Eingabemöglichkeiten unter Windows 8 geschuldet ist.

HINWEIS: Bitte erwarten Sie an dieser Stelle nicht, dass wir alle Controls in epischer Breite behandeln. Dafür ist die Online-Hilfe viel besser geeignet. Wir setzen an dieser Stelle voraus, dass Sie bereits mit den WPF-Controls in grundlegender Form vertraut sind (siehe dazu Kapitel 13).

19.1 Einfache WinRT-Controls

Zunächst beschäftigen wir uns mit einigen grundlegenden Controls, auf die wohl keine Anwendung verzichten kann.

HINWEIS: Die einzelnen Controls müssen Sie in eines der Layout-Controls (nur diese können mehrere Controls enthalten) einfügen. Bei den vorliegenden Seiten (*BlankPage*, *BasicPage*) finden Sie bereits ein zentrales *Grid* vor, das diese Aufgabe übernimmt. Weitere Informationen zu den Layout-Controls finden ab Seite 947.

19.1.1 TextBlock, RichTextBlock

Für die mehr oder weniger statische Textdarstellung eignen sich die Controls *TextBlock* bzw. *RichTextBlock*. Letzteres bietet zusätzlich die Möglichkeit, neben diverse Formatierungen (Schriftgröße, -farbe etc.) auch Grafiken darzustellen. Der *TextBlock* ist das Pendant bzw. der Ersatz für das bekannte WPF-Label-Control.

Einige kleine Beispiele sollen die Verwendung beider Controls demonstrieren.

BEISPIEL 19.1: Einfache Textausgabe mit vordefiniertem Style

XAML

```
...
   <TextBlock Text="TextBlock einfach" Style="{StaticResource BasicTextStyle}" />
...
```

BEISPIEL 19.2: *TextBlock* mit komplexerer Formatierung

XAML

```
...
```

Dieser *TextBlock* besteht aus einzelnen *Run*-Elementen, die wiederum unterschiedliche Inhalte, Formatierungen etc. aufweisen können. Der Vorteil: auf die einzelnen *Run*-Elemente kann auch per Name und damit zur Laufzeit per Code zugegriffen werden.

```
<TextBlock>
```

Dieses *Run*-Element ist an eine Eigenschaft der aktuellen Page gekoppelt:

```
<Run FontWeight="Bold" FontSize="24" Text="{Binding EineEigenschaft,
     ElementName=pageRoot}" />
```

Auch Zeilenumbrüche sind möglich:

```
<LineBreak/>
```

Unterschiedliche Formatierungen:

```
<Run FontWeight="Bold"  FontSize="24"
     Text="TextBlock mit Formatierung " />
<Run FontStyle="Italic" Foreground="Blue" Text="Es geht auch so!" />
```

Auch komplexe Füllpinsel lassen sich definieren:

```
<Run FontWeight="Bold" FontSize="35" Text="LinearGradientBrush">
    <Run.Foreground>
        <LinearGradientBrush>
            <GradientStop Color="Red" Offset="0.25" />
            <GradientStop Color="Green" Offset="0.5" />
        </LinearGradientBrush>
    </Run.Foreground>
</Run>
</TextBlock>
```

```
...
```

Ergebnis

Die Laufzeitansicht:

Im Gegensatz zum *TextBlock* bestehen beim *RichTextBlock* die einzelnen Absätze aus *Paragraph*-Elementen, diese wiederum können folgende Elemente enthalten: *Inline*, *InlineUIContainer*, *Run*, *Span*, *Bold*, *Italic*, *Underline*, *LineBreak*.

Einige Beispiele zeigen die Verwendung der einzelnen Elemente.

BEISPIEL 19.3: Verwendung *RichTextBlock*

XAML

```
...
            <RichTextBlock Name="rtb1" Margin="0,10,0,5" >
             <Paragraph FontSize="10">Absatz 1, Absatz 1, Absatz 1, Absatz 1, Absatz 1,
                  Absatz 1, Absatz 1, Absatz 1, Absatz 1</Paragraph>
```

Absatz mit erweiterten Formatierungen:

```
             <Paragraph FontSize="24" FontStretch="UltraExpanded" LineHeight="50"
                        CharacterSpacing="100" >Absatz 2, Absatz 2, Absatz 2, Absatz 2,
                        Absatz 2, Absatz 2, Absatz 2, Absatz 2, Absatz 2, Absatz 2,
                        Absatz 2</Paragraph>
```

Zeilenumbruch:

```
             <Paragraph><LineBreak/></Paragraph>
```

Schriftauszeichnungen festlegen:

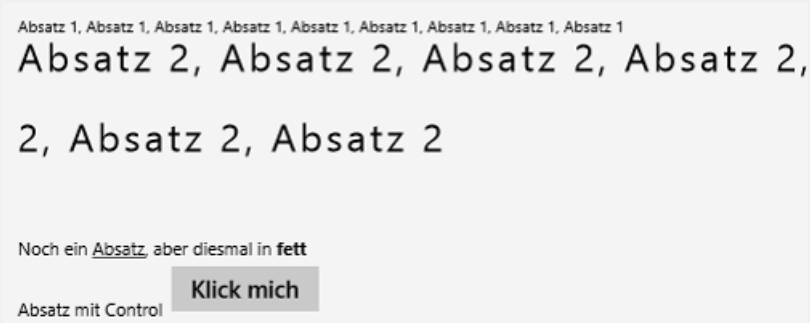
```
             <Paragraph>Noch ein <Underline>Absatz</Underline>, aber diesmal in
                   <Bold>fett</Bold>
             </Paragraph>
             <Paragraph>Absatz mit Control
```

Mittels *InlineUIContainer* wird ein *Button* im Text platziert:

```
                <InlineUIContainer>
                     <Button Click="Button_Click_1" Name="btn1">
                         Klick mich
                     </Button>
                </InlineUIContainer>
             </Paragraph>
          </RichTextBlock>
...
```

Ergebnis

Die Laufzeitansicht:



Die Schaltfläche ist natürlich auch funktionsfähig, auf das Abdrucken der Ereignisprozedur haben wir verzichtet.

Über das Schachteln von *Paragraph*-, *Run*- etc. Elementen können Sie auch zur Laufzeit Texte im *RichTextBlock* erzeugen. Entgegen der Erwartungen müssen Sie diese Elemente jedoch der Blocks-Auflistung übergeben.

BEISPIEL 19.4: Zeilen zur Laufzeit einfügen

C#
```
using Windows.UI.Xaml.Documents;
...
            Run run = new Run() { Text = "Text zur Laufzeit! Text zur Laufzeit!"};
            Paragraph para = new Paragraph();
            para.Inlines.Add(run);
            para.Inlines.Add(new LineBreak());
            para.Inlines.Add(new Run() { Text = "Noch mehr Text" });
            rtb2.Blocks.Add(para);
...
```

HINWEIS: Das *Hyperlink*-Element steht Ihnen hier, im Gegensatz zu WPF, nicht zur Verfügung, verwenden Sie stattdessen ein *InlineUIContainer*-Element und fügen Sie darin einen *HyperlinkButton* ein.

19.1.2 Button, HyperlinkButton, RepeatButton

Kommen wir nun zu den etwas "aktiveren" Steuerelementen. An dieser Stelle fassen wir mit *Button*, *HyperLinkButton* und *RepeatButton* drei recht ähnliche Controls zusammen. Allen gemein ist die Aufgabe, als Reaktion auf ein externes Ereignis eine Aktion auszuführen. Die Art des Ereignisses bestimmen Sie mit der Eigenschaft *ClickMode*:

- *Release*,
  das Ereignis tritt **nach** dem Klicken (Maustaste, Finger, Leertaste) auf.
- *Press*,
  das Ereignis tritt **beim** Klicken (Maustaste, Finger, Leertaste) auf.
- *Hover*,
  das Ereignis tritt ein, wenn Maustaste oder Finger über das Control bewegt werden.

BEISPIEL 19.5: Verwendung von Schaltflächen

XAML
```
...
            <Button>Einfacher Button</Button>
```

Schaltflächen können aus mehreren Controls bestehen:

```
            <Button>
                <StackPanel>
                    <Image Source="Images/mycomputer.png" Stretch="None"/>
                    <TextBlock HorizontalAlignment="Center">Button mit Grafik</TextBlock>
                </StackPanel>
            </Button>
```

BEISPIEL 19.5: Verwendung von Schaltflächen

```
            <HyperlinkButton Click="HyperlinkButton_Click_1">
                Ein HyperlinkButton
            </HyperlinkButton>
            <RepeatButton ClickMode="Hover" Click="RepeatButton_Click_1" >
                RepeatButton - Hover</RepeatButton>
            <RepeatButton ClickMode="Press" Click="RepeatButton_Click_1" >
                RepeatButton - Press</RepeatButton>
            <RepeatButton ClickMode="Release" Click="RepeatButton_Click_1" >
                RepeatButton - Release</RepeatButton>
...
```

C#

...

Die Funktionalität des *HyperlinkButton* muss erst programmiert werden:

```
async private void HyperlinkButton_Click_1(object sender, RoutedEventArgs e)
{
    Uri uri = new Uri(@"http://www.doko-buch.de");
    bool success = await Windows.System.Launcher.LaunchUriAsync(uri);
}
```

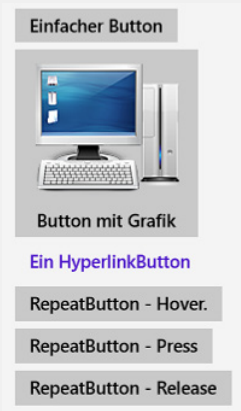
Die Reaktion der *RepeatButton*-Controls:

```
private void RepeatButton_Click_1(object sender, RoutedEventArgs e)
{
    (sender as RepeatButton).Content += ".";
}
```

...

Ergebnis

Die Laufzeitansicht:



HINWEIS: Es wird Ihnen sicher am Anfang etwas fremd sein, aber die meisten Ereignisse müssen Sie mit dem Schlüsselwort *async* deklarieren, denken Sie dabei an "Fast and Fluid" (siehe Seite 865).

19.1.3 CheckBox, RadioButton, ToggleButton, ToggleSwitch

Neben den beiden bekannten Controls *CheckBox* und *RadioButton*, die es dieser Form schon bei den Windows Forms gab, werden mit WinRT auch zwei neue Vertreter dieser Gattung eingeführt:

- *ToggleButton*
- *ToggleSwitch*

Gemein ist beiden, dass zwischen zwei Zuständen "geschaltet" werden kann. Beim *ToggleButton* wird dadurch die Eigenschaft *IsChecked* beeinflusst, beim *ToggleSwitch* die Eigenschaft *IsOn*.

BEISPIEL 19.6: Verwendung von *CheckBox, RadioButton, ToggleButton, ToggleSwitch*

XAML

```
...
       <StackPanel Grid.Row="1" Margin="120,0,120,0">
```

Zunächst ein paar *CheckBox*-Controls definieren:

```
           <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
               <CheckBox Margin="0,0,15,0">Option 1</CheckBox>
               <CheckBox IsChecked="True" Margin="0,0,15,0">Option 2</CheckBox>
               <CheckBox Margin="0,0,15,0">Option 3</CheckBox>
               <CheckBox Margin="0,0,15,0">Option 4</CheckBox>
```

Statt einer Beschriftung sind auch andere Inhalte denkbar:

```
               <CheckBox Margin="0,0,15,0" >
                   <Image Source="Images/cookie.png" Stretch="None"  />
               </CheckBox>
           </StackPanel>
           <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
```

Die zwei Zustände des *ToggleButton*s (auch hier könnten Sie Grafiken einblenden):

```
               <ToggleButton Margin="0,0,50,0">Das ist ein ToogleButton</ToggleButton>
               <ToggleButton IsChecked="True">Das ist ein ToogleButton</ToggleButton>
           </StackPanel>
           <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
```

Der *ToggleSwitch* in Aktion:

```
               <ToggleSwitch  Margin="0,0,50,0" IsOn="True">
                   Das ist ein ToggleSwitch</ToggleSwitch>
               <ToggleSwitch >Das ist ein ToggleSwitch</ToggleSwitch>
```

Alternativ kann den beiden Zuständen des "Schalters" auch unterschiedlicher Content zugewiesen werden (siehe Laufzeitansicht beim Klick auf den *ToggleSwitch*):

```
               <ToggleSwitch>
                   ToggleSwitch mit wechselnder Grafik
                   <ToggleSwitch.OnContent>
                       <Image Source="Images/bulbon.png"/>
                   </ToggleSwitch.OnContent>
```

BEISPIEL 19.6: Verwendung von *CheckBox, RadioButton, ToggleButton, ToggleSwitch*

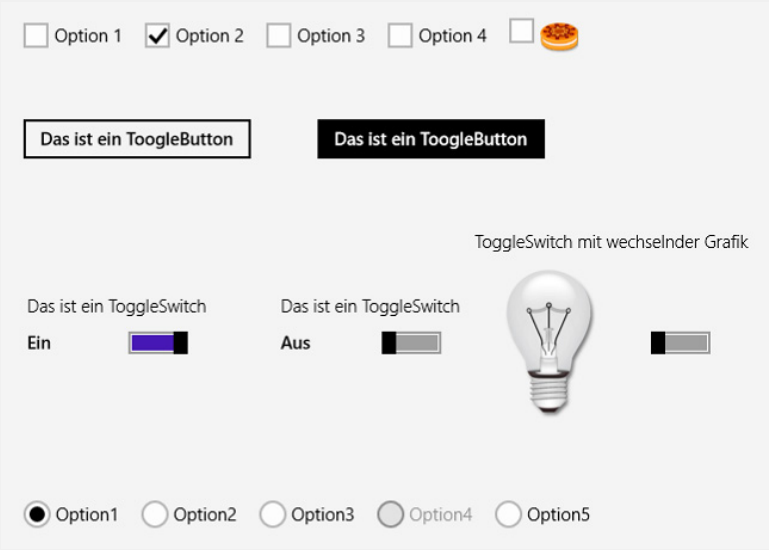
```
                    <ToggleSwitch.OffContent>
                        <Image Source="Images/bulboff.png"/>
                    </ToggleSwitch.OffContent>
                </ToggleSwitch>
            </StackPanel>
```

Noch ein paar *RadioButton*-Controls, diese müssen zum Gruppieren in einem übergeordneten Control zusammengefasst werden:

```
            <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
                <RadioButton IsChecked="True" Margin="0,0,15,0">Option1</RadioButton>
                <RadioButton Margin="0,0,15,0">Option2</RadioButton>
                <RadioButton Margin="0,0,15,0">Option3</RadioButton>
                <RadioButton IsEnabled="False" Margin="0,0,15,0">Option4</RadioButton>
                <RadioButton Margin="0,0,15,0">Option5</RadioButton>
            </StackPanel>
        </StackPanel>
...
```

Ergebnis

Die Laufzeitansicht:



19.1.4 TextBox, PasswordBox, RichEditBox

Für die Eingabe von Werten stehen Ihnen neben der bekannten *TextBox* noch die *RichEditBox* und der "Spezialist" *PasswordBox* zur Verfügung. Die Bedeutung der Controls dürfte sich aus dem Namen ergeben, einige Beispiele sollen die Verwendung demonstrieren.

BEISPIEL 19.7: *TextBox, PasswordBox, RichEditBox*

XAML

```
...
        <StackPanel Grid.Row="1" Margin="120,0,120,0">
            <TextBlock Text="TextBox einfach" Style="{StaticResource BasicTextStyle}"
                 Margin="0,10,0,5" />
```

Eine simple *TextBox* erzeugen:

```
            <TextBox Name="TextBox1"  Text="Eingabe"  />
            <Button Content="Alles Markieren" Click="Button_Click_3"/>
```

Eine mehrzeilige *TextBox* mit automatischem Zeilenumbruch erzeugen:

```
            <TextBox  Text="Eingabe"  Opacity="0.5"  Height="100"  BorderThickness="0"
                     Margin="0,10,0,5" AcceptsReturn="True" TextWrapping="Wrap" />
```

Passworteingabe leichtgemacht:

```
            <TextBlock Text="PasswordBox" Style="{StaticResource BasicTextStyle}"
                   Margin="0,10,0,5" />
            <PasswordBox  Password="abcdefg" PasswordChar="?" />
            <TextBlock Text="RichTextBox" Style="{StaticResource BasicTextStyle}"
                 Margin="0,10,0,5" />
```

Diese *RichEditBox* füllen wir zur Laufzeit:

```
            <RichEditBox  Height="200" Name="reb1"/>
```

Einige Schaltflächen, mit denen wir die *RichEditBox* manipulieren:

```
            <StackPanel Orientation="Horizontal">
                <Button Content="Laden" Click="Button_Click_1"/>
                <Button Content="Speichern" Click="Button_Click_4"/>
                <Button Content="Text Erzeugen" Click="Button_Click_2"/>
            </StackPanel>
...
```

C#

```
...
```

Wir laden eine Text- oder Richtext-Datei in die *RichEditBox*:

```
        async private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Das Öffnen der "Dateidialoge" – jetzt *FileOpenPicker* – erfordert die Vollbildansicht:

```
            if (ApplicationView.Value != ApplicationViewState.Snapped)
            {
                Windows.Storage.Pickers.FileOpenPicker open = new
                                    Windows.Storage.Pickers.FileOpenPicker();
                open.SuggestedStartLocation =
                       Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary;
                open.FileTypeFilter.Add(".rtf");
                open.FileTypeFilter.Add(".txt");
```

BEISPIEL 19.7: ***TextBox, PasswordBox, RichEditBox***

```
            Windows.Storage.StorageFile file = await open.PickSingleFileAsync();
```

Wurde eine Datei ausgewählt ...

```
            if (file != null)
            {
```

... können wir diese in die *RichEditBox* laden:

```
                Windows.Storage.Streams.IRandomAccessStream strm =
                    await file.OpenAsync(Windows.Storage.FileAccessMode.Read);
                reb1.Document.LoadFromStream(Windows.UI.Text.TextSetOptions.FormatRtf,
                                             strm);
            }
        }
    }
```

Alternativ können Sie die Inhalte der *RichEditBox* auch per Code erzeugen:

```
    async private void Button_Click_2(object sender, RoutedEventArgs e)
    {
```

Wie Sie sehen, arbeiten wir quasi mit der Einfügemarke, die wir verschieben:

```
        reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
```

Nachfolgend können wir Texte und Formatierungen zuweisen:

```
        reb1.Document.Selection.SetText(TextSetOptions.None, "Überschrift");
        reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.White;
        reb1.Document.Selection.CharacterFormat.Size = 18;

        reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
        reb1.Document.Selection.SetText(TextSetOptions.None, "Hallo in Grün");
        reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.Green;
        reb1.Document.Selection.CharacterFormat.Size = 10;

        reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
        reb1.Document.Selection.SetText(TextSetOptions.None, "Hallo in Rot");
        reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.Red;
```

Einfügen eines Hyperlinks:

```
        reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
```

Zunächst der Beschriftungstext:

```
        reb1.Document.Selection.SetText(TextSetOptions.None, "\nEin Hyperlink");
        reb1.Document.Selection.CharacterFormat.BackgroundColor = Colors.White;
        reb1.Document.Selection.CharacterFormat.ForegroundColor = Colors.Green;
```

Dann die Sprungadresse:

```
        reb1.Document.Selection.Link = "\"http://doko-buch.de\"";
```

BEISPIEL 19.7: *TextBox, PasswordBox, RichEditBox*

Hier fügen wir eine Grafik ein, die der Nutzer per *FileOpenPicker* auf dem System auswählt:

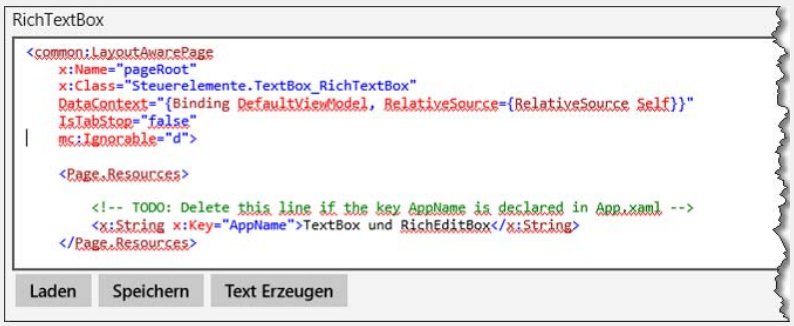
```
            Windows.Storage.Pickers.FileOpenPicker open = new
                            Windows.Storage.Pickers.FileOpenPicker();
            open.SuggestedStartLocation =
                    Windows.Storage.Pickers.PickerLocationId.PicturesLibrary;
            open.ViewMode = Windows.Storage.Pickers.PickerViewMode.Thumbnail;
            open.FileTypeFilter.Clear();
            open.FileTypeFilter.Add(".bmp");
            open.FileTypeFilter.Add(".png");
            open.FileTypeFilter.Add(".jpeg");
            open.FileTypeFilter.Add(".jpg");
            Windows.Storage.StorageFile file = await open.PickSingleFileAsync();
            Windows.Storage.Streams.IRandomAccessStream fileStream = await
                        file.OpenAsync(Windows.Storage.FileAccessMode.Read);

            reb1.Document.Selection.EndKey(TextRangeUnit.Story, false);
            reb1.Document.Selection.InsertImage(50, 50, 50,
                     VerticalCharacterAlignment.Top, "test", fileStream);
        }
```

Die Inhalte der *RichEditBox* speichern:

```
        async private void Button_Click_4(object sender, RoutedEventArgs e)
        {
            if (ApplicationView.Value != ApplicationViewState.Snapped)
            {
                Windows.Storage.Pickers.FileSavePicker save = new
                      Windows.Storage.Pickers.FileSavePicker();
                save.SuggestedStartLocation =
                     Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary;
                save.FileTypeChoices.Add("Rich Text Format", new String[] {".rtf"});
                Windows.Storage.StorageFile file = await save.PickSaveFileAsync();
                if (file != null)
                {
                    Windows.Storage.Streams.IRandomAccessStream strm =
                       await file.OpenAsync(Windows.Storage.FileAccessMode.ReadWrite);
                    reb1.Document.SaveToStream(TextGetOptions.FormatRtf, strm);
                    strm.Dispose();
                }
            }
        }
...
```

BEISPIEL 19.7: *TextBox, PasswordBox, RichEditBox*

Ergebnis



19.1.5 Image

Die Funktion dieses Controls dürfte sich wohl auf den ersten Blick erschließen, die per *Source*-Eigenschaft zugewiesenen Grafiken können in verschiedenen Ansichtsmodi (Eigenschaft *Stretch*) dargestellt werden.

HINWEIS: In .NET-Projekten wird die Darstellung von SVG-Grafiken **nicht** unterstützt. Sie müssen diese vorher mit einem Konverter in XAML-Grafiken umwandeln!

Das folgende Beispiel zeigt Ihnen neben der trivialen Anzeige von Grafiken auch die Möglichkeit, wie Sie diese per Gestensteuerung zur Laufzeit manipulieren können.

BEISPIEL 19.8: *Image*-Control und Gestensteuerung

XAML

...

Ein Ressourcen-Image in Originalgröße anzeigen:

```
<Image Source="Images/mycomputer.png" Stretch="None" Margin="0,0,0,20"/>
```

Das Bild wird in diesem Fall proportional auf 50x50 Pixel skaliert:

```
<Image Source="Images/mycomputer.png" Width="50" Height="50" Stretch="Uniform"
       Margin="0,0,0,20"/>
```

Das in den folgenden Canvas eingefügte *Image* wird für die Reaktion auf alle Manipulationsarten (Drehen, Vergrößern, Verschieben) konfiguriert. Die eigentliche Reaktion erfolgt im zugeordneten Eventhandler:

```
<Canvas Height="400" Background="AliceBlue">
   <Image Name="Image1" Source="Images/bulbon.png" ManipulationMode="All"
      ManipulationDelta="Image_ManipulationDelta"
           RenderTransformOrigin="0.5, 0.5">
       <Image.RenderTransform>
          <CompositeTransform/>
       </Image.RenderTransform>
```

BEISPIEL 19.8: *Image*-Control und Gestensteuerung

```
            </Image>
        </Canvas>
...
```

C#

Wie versprochen, müssen wir uns noch um die eigentliche Manipulation kümmern, wir hatten lediglich die Reaktion darauf "freigeschaltet":

```
private void Image_ManipulationDelta(object sender, ManipulationDeltaRoutedEventArgs e)
{
```

Aktuelle Einstellungen abrufen:

```
    CompositeTransform comptrans = (CompositeTransform)(sender as Image).RenderTransform;
```

Änderungen einrechnen:

```
    comptrans.ScaleX *= e.Delta.Scale;
    comptrans.ScaleY *= e.Delta.Scale;
    comptrans.TranslateX += e.Delta.Translation.X;
    comptrans.TranslateY += e.Delta.Translation.Y;
    comptrans.Rotation += 180.0 / Math.PI * e.Delta.Rotation;
}
...
```

Ergebnis

Testen Sie die Reaktion der "Glühlampe" auf die bekannten Gesten.

HINWEIS: Wer nicht über die erforderliche Hardware verfügt, der sei an den Tablet-Simulator verwiesen, hier können Sie die Gesten (Drehen, Verschieben, Vergrößern) über die drei entsprechenden Tasten auswählen und per Mausrad simulieren.

19.1.6 ScrollBar, Slider, ProgressBar, ProgressRing

An dieser Stelle fassen wir einige Controls zusammen, die ähnlich gelagerte Aufgaben übernehmen. Während *Scrollbar* und *Slider* für die Eingabe von Werten zwischen einem vorgegebenen *Minimum* und *Maximum* vorgesehen sind, stellt der *ProgressBar* die optische Anzeige eines Wertes dar. Eine Sonderrolle spielt der *ProgressRing*, dieser stellt einen neutralen Fortschritt oder eine Bearbeitung dar, die mit *IsActive* ein- oder ausgeschaltet wird.

Auf einige spezielle Eigenschaften des *Slider*-Controls geht das folgende Beispiel ein:

BEISPIEL 19.9: *Slider, ProgressBar, ProgressRing*

XAML

```
...
   <StackPanel Grid.Row="1" Margin="120,0,120,0">
            <StackPanel Orientation="Horizontal" Margin="0,50,0,50">
```

Ein horizontaler *Slider* mit einem Wertebereich von 1 bis 100:

```
               <Slider Name="Slider1" Width="300" Height="35" Minimum="1" Maximum="100"
                       Orientation="Horizontal" Value="25" Margin="0,0,50,0" />
```

Zur Anzeige binden wir einfach einen *TextBlock* an den *Value* des *Slider*s:

```
               <TextBlock Text="{Binding ElementName=Slider1, Path=Value}"
                          Style="{StaticResource HeaderTextStyle}" Width="75"/>
```

Ebenfalls an den *Slider* gebunden, ein *ProgressBar* zur Wertanzeige:

```
               <ProgressBar Maximum="100" Value="{Binding ElementName=Slider1, Path=Value}"
                       Margin="50,0,0,0" Height="25" Width="200" Foreground="GreenYellow"/>
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
               <TextBlock Style="{StaticResource TitleTextStyle}" >ProgressRing:</TextBlock>
               <CheckBox Name="Check2"  Margin="50,0,50,0">IsActive</CheckBox>
```

Ein *Slider* mit einer Schrittweite von 0,1 und einer Anzeigeschrittweite von 0,5:

```
            <StackPanel Orientation="Horizontal" Margin="0,25,0,20">
               <Slider Name="Slider2" Width="300" Height="35" Minimum="1" Maximum="5"
                       StepFrequency="0.1" SnapsTo="StepValues" TickFrequency="0.5"
                       Orientation="Horizontal" Value="25" Margin="0,0,50,0" />
               <TextBlock Text="{Binding ElementName=Slider2, Path=Value}"
                          Style="{StaticResource HeaderTextStyle}" />
```

Der regulär zurückgegebene Wert bestimmt sich aus der Eigenschaft *SnapsTo*. Ist diese auf *StepValue* festgelegt, wie im obigen Beispiel, wird der eingestellte Wert mit einer Genauigkeit von 0,1 bestimmt. Bei *TickValues* würden die Werte nur mit einer Schrittweite von 0,5 zurückgegeben werden.

```
            </StackPanel>
```

Ein *ProgressRing* zur Fortschrittsanzeige, das Ein-/Ausschalten übernimmt die *CheckBox*:

```
               <ProgressRing Height="60" Width="60" IsActive="{Binding IsChecked,
```

BEISPIEL 19.9: *Slider, ProgressBar, ProgressRing*

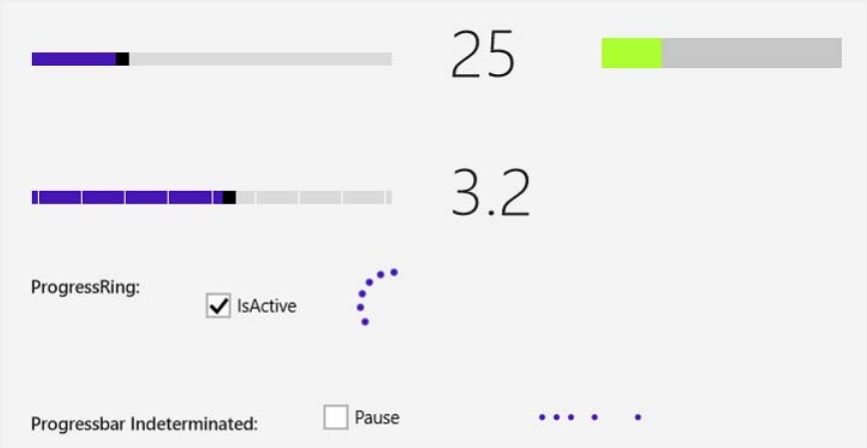
```
                              ElementName=Check2}"></ProgressRing>
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="0,50,0,0">
               <TextBlock Style="{StaticResource TitleTextStyle}"
                        VerticalAlignment="Center">Progressbar Indeterminated:</TextBlock>
               <CheckBox Name="Check1"  Margin="50,0,50,0">Pause</CheckBox>
```

Ein *ProgressBar* ohne absolute Wertanzeige (nur Animation):

```
               <ProgressBar Maximum="100" Value="{Binding ElementName=Slider1, Path=Value}"
                            IsIndeterminate="True" Height="25" Width="200"
                            Foreground="GreenYellow"
                            ShowPaused="{Binding IsChecked, ElementName=Check1}"/>
            </StackPanel>
...
```

Ergebnis

Verschieben Sie die beiden *Slider*:



19.1.7 Border, Ellipse, Rectangle

An dieser Stelle erwartet den WPF-Programmierer nichts Neues, die Controls verfügen lediglich über die zusätzlichen Events für die Toucheingabe (*Tapped* ...)

BEISPIEL 19.10: Verwendung *Border*

XAML

```
...
    <Border Height="100" Background="AliceBlue" BorderThickness="20,5,10,1"
            BorderBrush="Blue" CornerRadius="0,10,20,50" >
            <TextBlock Foreground="Blue" FontSize="50" HorizontalAlignment="Center"
                       VerticalAlignment="Center">Das ist ein Border</TextBlock>
   </Border>
...
```

BEISPIEL 19.10: Verwendung *Border*

Ergebnis

Die Laufzeitansicht:

19.2 Layout-Controls

Wie Sie den bisherigen Beispielen entnehmen konnten, kann in vielen Fällen immer nur ein Control als Content in den Clientbereich, z.B. einer *Page*, eingefügt werden. In diesem Fall können Sie auf den reichen Fundus an Layout-Controls zurückgreifen. Diese bieten nicht nur die Möglichkeit, mehrere Client-Controls aufzunehmen, sondern auch diese geschickt zu platzieren.

19.2.1 Canvas

Der *Canvas* erinnert noch am ehesten an das Erstellen von Windows Forms-Oberflächen, enthaltene Controls werden einfach über Ihre Koordinaten im Clientbereich positioniert.

HINWEIS: Beachten Sie, dass die Koordinaten per angehängter Eigenschaft zugewiesen werden (siehe folgende Beispiel).

BEISPIEL 19.11: Verwendung *Canvas*

C#

```
...
    <Canvas Height="200" Background="AntiqueWhite">
        <Image Canvas.Left="50" Canvas.Top="5" Source="Images/bulbon.png"/>
        <Image Canvas.Left="118" Canvas.Top="62" Source="Images/bulbon.png"/>
        <TextBlock Foreground="Blue" FontSize="24" Canvas.Left="118" Canvas.Top="51">
                  Canvas mit freier Positionierung</TextBlock>
    </Canvas>
...
```

Ergebnis

HINWEIS: Möchten Sie ein Control frei über anderen Controls positionieren, fügen Sie es in einen *Canvas* ein, dessen Größe Sie auf 0x0 Pixel reduzieren. Die *Canvas*-Größe hat keine Auswirkungen auf die Positionierbarkeit der enthaltenen Controls.

BEISPIEL 19.12: Frei positionierter *TextBlock*

XAML

```
...
        <!-- Back button and page title -->
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <Canvas Height="0" Width="0" Grid.Column="0" Grid.Row="0" >
                <TextBlock Foreground="Blue" FontSize="24" Canvas.Left="0"
                    Canvas.Top="-20">TextBlock mit freier Positionierung</TextBlock>
            </Canvas>
...
```

Ergebnis

19.2.2 StackPanel

Alle enthaltenen Controls werden in der Reihenfolge ihrer Definition "übereinandergestapelt". Standardmäßig erhalten die Controls die Breite des *StackPanel*s, Sie müssen also gegebenenfalls die Breite explizit festlegen. Mit der Eigenschaft *Orientation* legen Sie die Stapelrichtung (Horizontal, Vertikal) fest.

19.2.3 ScrollViewer

Kommen Sie mit den verfügbaren Platzverhältnissen nicht klar oder wollen Sie große Grafiken anzeigen, die nicht auf den Bildschirm passen, können Sie einen *ScrollViewer* einsetzen. Kann der enthaltene Content nicht komplett dargestellt werden, blendet der *ScrollViewer* horizontale und/oder vertikale Scrollbars ein, die das Verschieben des Sichtausschnitts ermöglichen.

BEISPIEL 19.13: Verwendung *ScrollViewer*

XAML

```
...
      <ScrollViewer Height="200" Width="200" HorizontalScrollBarVisibility="Auto" >
        <Image Source="Images/bulbon.png" Width="400" Height="400" />
       </ScrollViewer>
...
```

BEISPIEL 19.13: Verwendung *ScrollViewer*

Ergebnis

19.2.4 Grid

Das wohl wichtigste Layout-Control ist das *Grid*. Ein Blick in den XAML-Code, z.B. einer Standardseite, zeigt recht schnell, dass ohne dieses Control nichts läuft. Wie bei einer Tabelle werden zunächst Zeilen und Spalten definiert. Diese können absolute Angaben (in Pixeln) zu Höhe und Breite enthalten oder auch relative (siehe Beispiel). Alternativ bietet es sich auch an, dass der Inhalt die Höhe bzw. die Breite bestimmt, in diesem Fall wird für *Height* oder *Width* der Wert "Auto" übergeben. Die enthaltenen Controls lassen sich mittels der angehängten Eigenschaften *Grid.Column* und *Grid.Row* im *Grid* positionieren.

BEISPIEL 19.14: Verwendung *Grid*

XAML

...

Zunächst die Maße des *Grid*s bestimmen und die Zeilen definieren:

```
<Grid Height="200" Width="500" Background="LightYellow" >
   <Grid.RowDefinitions>
       <RowDefinition Height="Auto"/>
       <RowDefinition Height="3*"/>
       <RowDefinition Height="2*"/>
   </Grid.RowDefinitions>
```

Die Höhe der ersten Zeile wird durch den Inhalt bestimmt, den Rest teilen sich Zeile 2 und 3 im Verhältnis 3 zu 2.

Die Spalten festlegen:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="3*"/>
    <ColumnDefinition Width="1*"/>
</Grid.ColumnDefinitions>
```

Die Breite der ersten Spalte wird durch den Inhalt bestimmt, den Rest teilen sich Spalte 2 und 3 im Verhältnis 3 zu 1.

BEISPIEL 19.14: Verwendung *Grid*

Die folgenden drei Rechtecke werden in den Gridzellen positioniert:

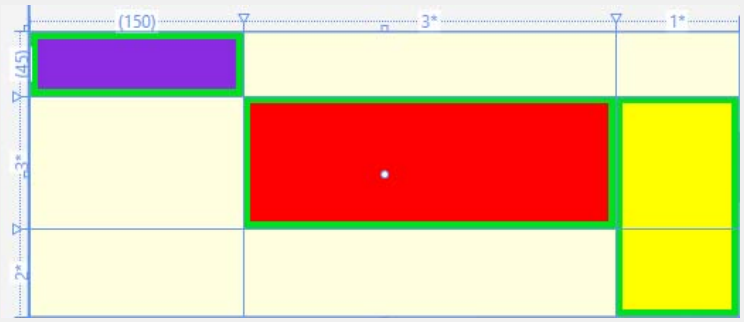
```
<Rectangle Grid.Row="0" Grid.Column="0" Width="150" Height="45"
 Fill="BlueViolet" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
<Rectangle Grid.Row="1" Grid.Column="1" Fill="Red" StrokeThickness="5"
 Stroke="#FF00E01F" ></Rectangle>
<Rectangle Grid.Row="1" Grid.Column="2" Grid.RowSpan="2" Fill="Yellow"
           StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Beachten Sie das dritte Rechteck: Dieses erstreckt sich wegen *Grid.RowSpan="2"* über zwei Zeilen.

```
            </Grid>
...
```

Ergebnis

Die Entwurfsansicht:



19.2.5 VariableSizedWrapGrid

Bei diesem erweiterten *WrapGrid* bestimmen Sie zunächst mit *Orientation* die Ausrichtung (Standard *Vertical*) und die Standardhöhe und -breite der enthaltenen Items (*ItemHeight*, *ItemWidth*). Die Eigenschaft *MaximumRowsOrColumns* legt fest, nach welchem Element umgebrochen wird (abhängig von *Orientation*). Die enthaltenen Elemente werden auf das mit *ItemHeight* und *ItemWidth* festgelegte Maß beschnitten, es sei denn, Sie weisen dem Element eine *ColumnSpan* oder *RowSpan* zu (das Element belegt dann mehr als eine Spalte oder Zeile).

BEISPIEL 19.15: Verwendung *VariableSizedWrapGrid*

XAML

VariableSizedWrapGrid mit drei Zeilen und einer Zellgröße von 50x50 Pixeln definieren:

```
...
   <VariableSizedWrapGrid Height="160" Background="LavenderBlush" ItemHeight="50"
                          ItemWidth="50" MaximumRowsOrColumns="3">
```

Das folgende Rechteck ist kleiner als der "Cell"-Bereich und wird original angezeigt:

```
      <Rectangle Fill="PaleGreen" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

BEISPIEL 19.15: Verwendung *VariableSizedWrapGrid*

```
        <Rectangle Fill="DarkRed" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Ein Rechteck mit doppelter Höhe:

```
        <Rectangle Fill="Red" StrokeThickness="5" Stroke="#FF00E01F" Height="100"
                   VariableSizedWrapGrid.RowSpan="2" ></Rectangle>
        <Rectangle Fill="Green" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Ein Rechteck mit doppelter Breite:

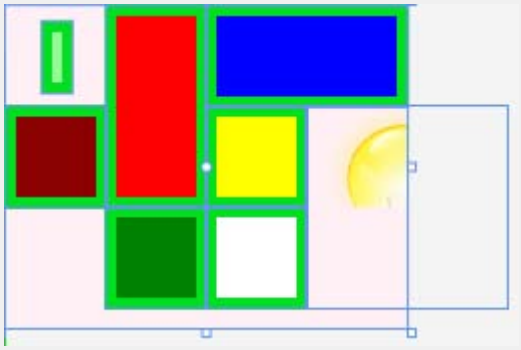
```
        <Rectangle Fill="Blue" StrokeThickness="5" Stroke="#FF00E01F" Width="100"
                   VariableSizedWrapGrid.ColumnSpan="2" ></Rectangle>
        <Rectangle Fill="Yellow" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
        <Rectangle Fill="White" StrokeThickness="5" Stroke="#FF00E01F" ></Rectangle>
```

Das Image wird beschnitten, da zu groß:

```
        <Image Source="Images/bulbon.png" Height="100" Width="100"/>
    </VariableSizedWrapGrid>
...
```

Ergebnis

Die Entwurfsansicht:



19.3 Listendarstellungen

Im Zusammenhang mit der Anzeige von Datenlisten etc. werden auch entsprechende Controls benötigt, die diese Listen auf sinnvolle Weise visualisieren können.

19.3.1 ComboBox, ListBox

Eigentlich sollten Sie sich im Zusammenhang mit WinRT-Anwendungen von der guten alten *ComboBox* verabschieden, ist diese doch für die Touchbedienung nur noch eingeschränkt empfehlenswert. Aber wer trennt sich schon gern von lieb gewonnenen Lösungen.

Beide Controls basieren auf einem *ItemsControl,* können also eine Liste von einzelnen *Items* aufnehmen.

BEISPIEL 19.16: Einfache *ListBox*, die Items werden per XAML definiert

XAML

```
...
            <ListBox>
                <ListBoxItem>Zeile 1</ListBoxItem>
                <x:String>Eintrag 2</x:String>
                <x:String>Eintrag 3</x:String>
                <x:String>Eintrag 4</x:String>
                <Image Source="Images/bulbon.png" Width="40"></Image>
            </ListBox>
...
```

Ergebnis

Wie Sie sehen, kann es sich um recht unterschiedliche Objekte handeln, die Sie sowohl in einer *ListBox* als auch in einer *ComboBox* unterbringen können.

Doch was ist eigentlich mit Daten in der einfachen Form einer *String*-Liste?

BEISPIEL 19.17: Datenbindung *ListBox*

C#

```
...
    public sealed partial class Listen : Steuerelemente.Common.LayoutAwarePage
    {
```

Wir definieren die Liste als Eigenschaft der aktuellen Seite:

```
        public List<string> Items { get; set; }
```

Im Konstruktor weisen wir der Liste einige Werte zu:

```
        public Listen()
        {
            this.InitializeComponent();
            Items = new List<string>();
            for (int i = 0; i < 100; i++)
                Items.Add("Eintrag " + i.ToString());
        }
...
```

XAML

```
<common:LayoutAwarePage
    x:Name="pageRoot"
...
   <ListBox ItemsSource="{Binding Items, ElementName=pageRoot}" Width="200"/>
...
```

Wie Sie sehen, genügt jetzt das Binden an die aktuelle Seite (*ElementName=pageRoot*), die eigentlich zu bindende Eigenschaft *Items* können Sie wie oben gezeigt oder per *Binding Path=Items* bestimmen.

Die gleiche Vorgehensweise ist auch bei der *ComboBox* relevant, allerdings ist deren Erscheinung an die Gegebenheiten einer Touchoberfläche angepasst worden:

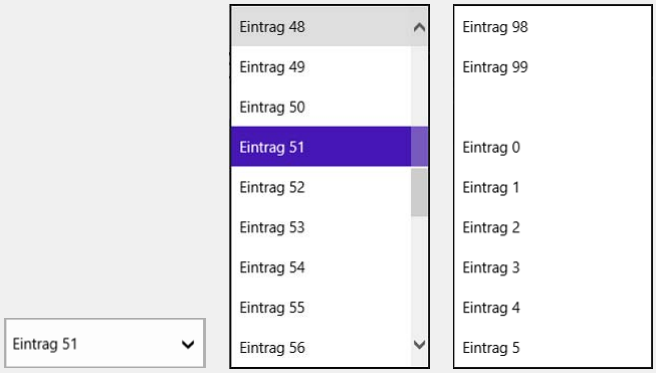
BEISPIEL 19.18: Verwendung *ComboBox*

XAML

```
...
   <ComboBox ItemsSource="{Binding Path=Items, ElementName=pageRoot}" Width="200"
             Height="50" VerticalAlignment="Top"/>
...
```

Ergebnis

Die Laufzeitansicht vor und nach dem Klick:



Um eine bessere Touchbedienung zu erreichen, wird die Auswahlliste möglichst zentriert über der *ComboBox* angezeigt. Wie die dritte Abbildung zeigt, kann dabei auch das Listenende "überschritten" werden, die Liste beginnt dann wieder von vorn.

Eine etwas kompliziertere Variante der Datenbindung zeigt das folgende Beispiel. Ausgehend von der Klasse *Artikel (Name, Preis)* soll eine Liste dieser Daten in einer *ListBox* dargestellt werden.

BEISPIEL 19.19: Binden an eine *ObservableCollection*

C#

Zunächst die Klasse *Artikel* definieren (wir implementieren *INotifyPropertyChanged*, um auf Eigenschaftsänderungen reagieren zu können):

```
...
    public class Artikel : INotifyPropertyChanged
    {
```

Die beiden Eigenschaften Name und Preis:

```
        private string _name;
        public string Name
        {
```

BEISPIEL 19.19: Binden an eine *ObservableCollection*

```
            get { return _name; }
            set
            {
                _name = value;
                this.OnPropertyChanged("Name");
            }
        }

        private double _preis;
        public double Preis
        {
            get { return _preis; }
            set
            {
                _preis = value;
                this.OnPropertyChanged("Preis");
            }
        }

        public Artikel()
        { }
```

Die statische Methode *GetBeispielArtikel* liefert uns eine bereits gefüllte Liste von Artikeln:

```
        public static ObservableCollection<Artikel> GetBeispielArtikel()
        {
            ObservableCollection<Artikel> artikel = new ObservableCollection<Artikel>()
        {
            new Artikel() {Name = "Bier", Preis= 1.20},
            new Artikel() {Name = "Wein", Preis= 3.70},
...
            new Artikel() {Name = "Kohlrabi", Preis= 0.50}
        };
            return artikel;
        }
```

Die Implementierung des *PropertyChanged*-Ereignisses:

```
        public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
        protected virtual void OnPropertyChanged(string propertyName)
        {
            if (this.PropertyChanged != null)
            {
                this.PropertyChanged(this,
                  new System.ComponentModel.PropertyChangedEventArgs(propertyName));
            }
        }
    }
...
```

BEISPIEL 19.19: Binden an eine *ObservableCollection*

In der eigentlichen Seite definieren wir zunächst eine Eigenschaft vom Typ *ObservableCollection<Artikel>* und weisen dieser die Beispieldaten zu:

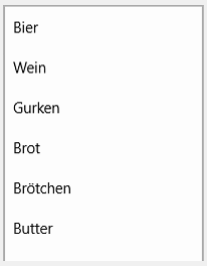
```
...
        public ObservableCollection<Artikel> Artikeliste { get; set; }

        public Listen()
        {
            this.InitializeComponent();
            Artikeliste = Artikel.GetBeispielArtikel();
...
```

XAML

Die Datenquelle wird in diesem Fall mit der Eigenschaft *Artikelliste* zugewiesen, der anzuzeigende Member ist das Feld *Name*:

```
...
  <ListBox ItemsSource="{Binding Artikeliste, ElementName=pageRoot}"
           DisplayMemberPath="Name" Width="200"/>
...
```

Ergebnis



19.3.2 ListView

Soll neben dem Namen des Artikels auch der Preis oder z.B. eine Abbildung etc. angezeigt werden, bietet es sich an, statt der *ListBox* gleich eine *ListView* zu verwenden. Prinzipiell sind beide Controls sehr ähnlich, die *ListView* ist aber die etwas komplexere Variante, die vor allem einige neue Ereignisse (*DragItemsStarting*, *ItemsClick*) und Unterstützung für SemanticZoom und IncrementalLoading bietet.

BEISPIEL 19.20: Verwendung *ListView*

XAML

```
...
    <ListView ItemsSource="{Binding Artikeliste, ElementName=pageRoot}" Width="150"
        Height="450" VerticalAlignment="Top" SelectionMode="Extended" Header="Artikel">
```

Wie Sie sehen, binden wir die *ListView* ebenfalls an die schon aus dem vorhergehenden Beispiel 19.19 bekannte Artikelliste.

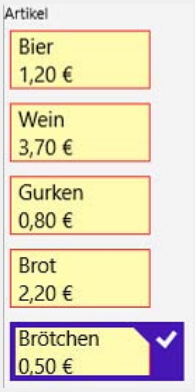
In diesem Fall wollen wir allerdings nicht nur den Namen sondern auch den Preis anzeigen lassen. Dazu nutzen wir ein *DataTemplate*, das uns freie Gestaltungsmöglichkeiten bietet:

BEISPIEL 19.20: Verwendung *ListView*

```
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <Border Background="#FFFFFBB1" Padding="5,0,5,0"
                                BorderBrush="#FFF93131" BorderThickness="1" Width="100">
                            <StackPanel>
                                <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"/>
                                <TextBlock Text="{Binding Preis,
                                  ConverterParameter='{}{0:C}',
                                  Converter={StaticResource StringFormatConverter1}}"
                                  HorizontalAlignment="Left"/>
                            </StackPanel>
                        </Border>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
...
```

Ergebnis

Die Laufzeitansicht (der Scrollbar ist nur sichtbar, wenn die *ListView* den Fokus besitzt):



Der im Beispiel verwendete *StringFormatConverter* wird im Abschnitt 19.5.1 ab Seite 967 vorgestellt.

HINWEIS: Wer bereits in WPF mit der *ListView* gearbeitet hat, wird sich vielleicht an die Möglichkeit erinnern, mehrere Spalten innerhalb der *ListView* zu generieren und diesen die jeweiligen Member der Collection zuzuordnen. Dies ist bei der WinRT-Variante nicht möglich!

19.3.3 GridView

Möchten Sie die Daten nicht nur in einer Spalte sondern auch mehrspaltig anzeigen, können Sie ein *GridView* verwenden.

BEISPIEL 19.21: Verwendung *GridView*

XAML

Wir binden die *GridView* an die aus dem Beispiel 20.19 bekannte Artikelliste.

```
...
            <StackPanel>
```

Mit dem Klick auf einen Item zeigen wir in folgendem *TextBlock* den Inhalt an:

```
                <TextBlock Name="txt1" Style="{StaticResource TitleTextStyle}"
                           Margin="5,0,0,0"/>
```

Die *GridView*:

```
                <GridView ItemsSource="{Binding Artikeliste, ElementName=pageRoot}"
                          Width="400" SelectionMode="Extended" VerticalAlignment="Top"
                          ItemClick="GridView_ItemClick_1" IsItemClickEnabled="True">
```

Das *DataTemplate* definiert das Aussehen der einzelnen Items:

```
                    <GridView.ItemTemplate>
                        <DataTemplate>
                            <Border Background="#FFFFFBB1" Width="150" Height="80"
                                    Padding="5,0,5,5" BorderBrush="#FFF93131">
                                <StackPanel>
                                    <TextBlock Text="{Binding Name}"
                                               HorizontalAlignment="Left"/>
                                    <TextBlock Text="{Binding Preis, ConverterParameter='{}
                                     {0:C}', Converter={StaticResource
                                       StringFormatConverter1}}" HorizontalAlignment="Left"/>
                                    <TextBox Text="{Binding Name, Mode=TwoWay}" />
                                </StackPanel>
                            </Border>
                        </DataTemplate>
                    </GridView.ItemTemplate>
```

Die Anzahl der maximal angezeigten Zeilen festlegen:

```
                    <GridView.ItemsPanel>
                        <ItemsPanelTemplate>
                            <WrapGrid MaximumRowsOrColumns="5"
                                  VerticalChildrenAlignment="Top"
                                  HorizontalChildrenAlignment="Left"/>
                        </ItemsPanelTemplate>
                    </GridView.ItemsPanel>
                </GridView>
            </StackPanel>
...
```

BEISPIEL 19.21: Verwendung *GridView*

C#

Über das *ItemClick*-Ereignis (dieses muss mit *IsItemClickEnabled* freigeschaltet werden) können wir den aktuell gewählten Item auswerten (Typisierung nicht vergessen):

```
...
        private void GridView_ItemClick_1(object sender, ItemClickEventArgs e)
        {
            Artikel item = e.ClickedItem as Artikel;
            txt1.Text = item.Name;
        }
...
```

Ergebnis

Die Laufzeitansicht (der Scrollbar ist nur sichtbar, wenn das *GridView* den Fokus besitzt):

Da wir in diesem Fall auch eine *TextBox* im *DataTemplate* eingeblendet haben, können wir die Daten natürlich auch bearbeiten, was wiederum Auswirkungen auf die anderen gebundenen Controls auf der aktuellen Seite hat:

HINWEIS: Weitere Anwendungsbeispiele für das *GridView* finden Sie im Abschnitt 19.5 ab Seite 967.

19.3.4 FlipView

Der eine oder andere wird sich vielleicht wundern, was das *FlipView*-Control an dieser Stelle zu suchen hat, aber es handelt sich (wie bei den vorhergehenden Controls auch) um ein von *ItemsControl* abgeleitetes Control. Damit hat es die Fähigkeit, weitere Items aufzunehmen.

Die sicher plausibelste Verwendung ist die Anzeige mehrerer Grafiken, zwischen denen mit Hilfe zweier Schaltflächen gewechselt werden kann.

BEISPIEL 19.22: Verwendung *FlipView* für Bilder

XAML

```
...
          <FlipView  Name="flv1" MaxHeight="200" MaxWidth="400"
                    SelectionChanged="FlipView_SelectionChanged">
              <Image Source="Images/bulboff.png" />
              <Image Source="Images/bulbon.png" />
              <Image Source="Images/mycomputer.png" />
          </FlipView>
...
```

Ergebnis

Laufzeitansicht (die Schaltflächen sind nur sichtbar, wenn die *FlipView* den Fokus besitzt):

Möchten Sie ein größeres Image (z.B. 1:1-Ansicht) an obige *FlipView* binden, funktioniert dies wie folgt:

```
<Image Source="{Binding ElementName=flv1, Path=SelectedItem.Source}" Stretch="None"
       Margin="0,50,0,50"/>
```

Doch die Anzeige und Auswahl von Bildern ist nicht die einzige Verwendungsmöglichkeit. Es ist ebenfalls problemlos möglich, die *FlipView* an eine Collection zu binden und mittels *DataTemplate* die Daten formatiert anzuzeigen:

BEISPIEL 19.23: Verwendung *ListView* mit Datenbindung

XAML

```
...
          <FlipView Width="400" Height="100" ItemsSource="{Binding Artikeliste,
                    ElementName=pageRoot}" Background="AliceBlue">
              <FlipView.ItemTemplate>
                  <DataTemplate>
                     <StackPanel Margin="120,0,120,0">
                          <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"
```

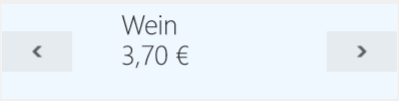
BEISPIEL 19.23: Verwendung *ListView* mit Datenbindung

```
                                Style="{StaticResource SubheaderTextStyle}" />
                            <TextBlock Text="{Binding Preis, ConverterParameter='{}{0:C}',
                                Converter={StaticResource StringFormatConverter}}"
                                HorizontalAlignment="Left"
                                Style="{StaticResource SubheaderTextStyle}"/>
                        </StackPanel>
                    </DataTemplate>
                </FlipView.ItemTemplate>
            </FlipView>
...
```

Wie Sie sehen, binden wir die *FlipView* ebenfalls an die schon aus dem Beispiel 19.19 bekannte Artikelliste.

Ergebnis

Laufzeitansicht (die Schaltflächen sind nur sichtbar, wenn die *FlipView* den Fokus besitzt):



HINWEIS: Ob obige Lösung auch intuitiv ist sei dahingestellt. Im Normalfall sieht der User ja nur die reinen Daten und nicht die Navigationsschaltflächen, ob es sich um eine Auswahl/Navigation handelt, muss man dann erst "ausprobieren".

Möchten Sie die Navigationsschaltflächen am oberen/unteren Rand einblenden, verwenden Sie einfach folgendes *ItemsPanel*:

```
            <FlipView Width="400" Height="200" ItemsSource="{Binding Artikeliste,
                      ElementName=pageRoot}" Background="AliceBlue">
                <FlipView.ItemTemplate>
                    <DataTemplate>
....
                    </DataTemplate>
                </FlipView.ItemTemplate>
                <FlipView.ItemsPanel>
                    <ItemsPanelTemplate>
                        <StackPanel Background="Transparent" Orientation="Vertical"/>
                    </ItemsPanelTemplate>
                </FlipView.ItemsPanel>
            </FlipView>
```

19.4 Sonstige Controls

An dieser Stelle wollen wir uns einigen mehr oder weniger neuen Controls zuwenden, die aber für das Erstellen von multimedialen Oberflächen unabdingbar sind.

19.4.1 CaptureElement

Geht es um die Anzeige der Daten einer Webcam (Tablets werden meist über eine Front- und eine Rückseiten-Kamera verfügen), bietet sich für die reine Anzeige ein *CaptureElement* an.

Weisen Sie dazu der *Source*-Eigenschaft zur Laufzeit ein initialisiertes *MediaCapture*-Objekt zu. Was sich so einfach anhört, erfordert allerdings einigen Zusatzaufwand, wie es das folgende kleine Beispiel zeigt:

BEISPIEL 19.24: Verwendung *CaptureElement*

XAML

```
...
       <StackPanel Grid.Row="1" Margin="120,0,120,0">
           <CaptureElement Width="320" Height="240" Margin="0,0,0,30"
                           Name="CaptureElement1"/>
           <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
               <Button Click="Button_Click_1" Width="100">Start</Button>
               <Button Click="Button_Click_3" Width="100">Stop</Button>
               <Button Click="Button_Click_2" Width="100">Drehen</Button>
           </StackPanel>
       </StackPanel>
...
```

C#

```
...
using Windows.Devices.Enumeration;
using Windows.Media.Capture;
...
    public sealed partial class Capture_Beispiel : Steuerelemente.Common.LayoutAwarePage
    {
```

Eine Instanz der *MediaCapture*-Klasse definieren:

```
MediaCapture medcap;
```

Mit dem Laden der Seite bestimmen wir zunächst die Anzahl der Video-Kameras und wählen dann das erste Gerät in der Liste aus. Alternativ können Sie über die *DeviceInformationCollection* auch Informationen über die einzelnen Geräte auswerten bevor Sie eines zuweisen:

```
private async void pageRoot_Loaded(object sender, RoutedEventArgs e)
{
    DeviceInformationCollection videocams = await
             DeviceInformation.FindAllAsync(DeviceClass.VideoCapture);
```

Ist mindestens ein Videogerät vorhanden, wählen wir dieses aus und initialisieren sowohl die *MediaCapture*-Instanz als auch das *CaptureElement*:

BEISPIEL 19.24: Verwendung *CaptureElement*

```
            if (videocams.Count > 0)
            {
                medcap = new MediaCapture();
                MediaCaptureInitializationSettings mcis = new
                       MediaCaptureInitializationSettings();
                mcis.VideoDeviceId = videocams[0].Id;
                await medcap.InitializeAsync( );
                CaptureElement1.Source = medcap;
            }
        }
```

Zu diesem Zeitpunkt wird noch nichts angezeigt, dies wird erst mit dem Aufruf der *StartPreviewAsync*-Methode realisiert:

```
        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            await medcap.StartPreviewAsync();
        }
```

Stoppen der Anzeige:

```
        private async void Button_Click_3(object sender, RoutedEventArgs e)
        {
            await medcap.StopPreviewAsync();
        }
```

Für die Art und Weise der Darstellung ist nicht das *CaptureElement* zuständig sondern die Datenquelle, d h. das *MediaCapture*-Objekt (in diesem Fall drehen wir die Anzeige):

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            medcap.SetPreviewRotation(VideoRotation.Clockwise180Degrees);
        }
    }
}
```

HINWEIS: Wenn Sie die App starten, dürfen Sie nicht vergessen, vorher die erforderlichen Funktionen in der *Package.appxmanifest*-Datei entsprechend zu setzen. Wichtig: Sie müssen Webcam **und** Mikrofon anfordern, andernfalls kommt es zu einer Fehlermeldung, auch wenn Sie nur auf das Video zugreifen wollen. Nach dem Start wird dem Anwender zunächst die obligatorische Sicherheitsabfrage (Zugriff Webcam/Mikrofon) angezeigt.

19.4.2 MediaElement

Wie es sein Name schon vermuten lässt, können Sie das *MediaElement* für die Wiedergabe von Video- oder Musik-Dateien/-Streams nutzen. Verwenden Sie die *Source*-Eigenschaft, um den

gewünschten Dateinamen anzugeben, alternativ können Sie auch zur Laufzeit mit der *SetSource*-Methode einen Stream übergeben.

Die wesentlichsten Methoden zur Steuerung der Wiedergabe sind:

- *Play*
- *Stop*
- *Pause*

HINWEIS: Haben Sie die Eigenschaft *AutoPlay* auf *True* gesetzt, startet die Wiedergabe nach dem Zuweisen der *Source*-Eigenschaft (bzw. wenn diese bereits gesetzt ist) mit der Anzeige des Controls.

BEISPIEL 19.25: Laden eines Videos aus dem aktuellen App-Unterverzeichnis *videos*

C#
```
...
        private async void Button_Click_4(object sender, RoutedEventArgs e)
        {
            StorageFolder folder = await
                                Package.Current.InstalledLocation.GetFolderAsync("videos");
            StorageFile file = await folder.GetFileAsync("video2.mts");
            var stream = await file.OpenAsync(FileAccessMode.Read);
            MediaElement1.SetSource(stream, file.ContentType);
        }
...
```

Über die Ereignisse *MediaOpened*, *MediaEnded* und *CurrentStateChanged* können Sie den aktuellen Status der Wiedergabe auswerten und gegebenenfalls eine neue Datei laden bzw. den Status der Steuerungstasten anpassen.

Für die Endloswiedergabe nutzen Sie *IsLooping*, *IsMuted* sorgt für Ruhe. Über *Volume* steuern Sie die Lautstärke.

BEISPIEL 19.26: Lautstärkeregelung mit Datenbindung eines *Sliders*

XAML
```
...
    <Slider Maximum="1" Minimum="0" Value="{Binding ElementName=Media1, Path=Volume,
            Mode=TwoWay}" Width="200" TickFrequency=".01" StepFrequency="0.01"
            SnapsTo="Ticks" Margin="30,0,0,0"/>
...
```

HINWEIS: Die aktuelle Wiedergabeposition können Sie nur per *MediaElement.Position.TotalSeconds*-Eigenschaft und zusätzlichen *Timer* auslesen – ein Ereignis mit entsprechendem Parameter gibt es dafür **nicht**. Alternativ lässt sich jedoch zum Beispiel ein *ProgressBar* direkt an ein *MediaElement* und damit auch an dessen *Position*-Eigenschaft binden.

BEISPIEL 19.27: Fortschrittsanzeige mittels *ProgressBar* realisieren

XAML
```
...
<MediaElement Name="Media1" />
<ProgressBar  Width="300"  Margin="0,10,0,0"
        Maximum="{Binding ElementName=Media1, Path=NaturalDuration.TimeSpan.TotalSeconds}"
       Value="{Binding ElementName=Media1, Path=Position.TotalSeconds}"/>
...
```

Über die unterstützten Dateiformate können Sie sich unter folgender Adresse informieren:

LINK: http://msdn.microsoft.com/en-us/library/hh986969.aspx

Mit einem etwas speziellerem Thema beschäftigt sich das Praxisbeispiel 19.5.7 "Musikwiedergabe im Hintergrund realisieren" ab Seite 988.

19.4.3 Frame

Im Abschnitt 18.1.2 (Die Page, der Frame und das Window) ab Seite 901 haben wir Ihnen bereits das organisatorische Grundkonzept der WinRT-Apps aufgezeigt. Ausgangspunkt der App war das einzige Fenster der App, als dessen Inhalt ein zentraler Frame zugewiesen wurde. Alle Seiten Ihrer WinRT-App werden innerhalb dieses Frames dargestellt, der Frame ist also nur ein Container für die eigentlichen Inhalte. Auch die Navigation zwischen den einzelnen Seiten ist Aufgabe dieses Frames (siehe dazu Abschnitt 18.3 ab Seite 915).

Was an so zentraler Stelle funktioniert, lässt sich aber auch in den einzelnen Seiten nutzen, um zum Beispiel schnell Inhalte innerhalb einer Page zu organisieren, siehe dazu das Praxisbeispiel 18.6.2 "Auf Ansichtsänderungen reagieren" ab Seite 926.

19.4.4 WebView

Für die **einfache** Anzeige von Webseiten und auch HTML-Strings bietet sich das *WebView*-Control an. Wer jetzt an das WPF *Browser*-Control denkt, liegt teilweise falsch. Auch wenn es die beiden Methoden *Navigate* (Aufruf URL) bzw. *NavigateToString* (Anzeige HTML-Quelltext inklusive Skripts) ebenfalls gibt – der wesentlichste Unterschied ist die mangelnde Unterstützung für Navigations-Ereignisse bzw. die gänzlich fehlende Unterstützung für die Seitennavigation.

BEISPIEL 19.28: Anzeige Webseite in einer *WebView*

C#
```
...
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            Uri uri = new Uri(@"http://www.doko-buch.de");
            WebView1.Navigate(uri);
        }
...
```

BEISPIEL 19.29: Anzeige HTML-Code in einer WebView

C#
```
...
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
           WebView1.NavigateToString("<html><body><b>Hier könnte Ihr HTML-Text stehen
                   ...</b></body></html>");
        }
...
```

Einziges verwertbares Feedback des Controls ist neben dem *NavigationFailed* das *LoadCompleted*-Ereignis:

BEISPIEL 19.30: Auswerten des *LoadCompleted*-Ereignisses

C#
```
...
        private void pageRoot_Loaded_1(object sender, RoutedEventArgs e)
        {
            WebView1.LoadCompleted += WebView1_LoadCompleted;
        }
```

Wir zeigen nach dem Laden der Webseite eine kleine Vorschau in einem *Rectangle*-Control an. Dazu nutzen wir einen *WebViewBrush*, der die gerenderte Webseite enthält:

```
        void WebView1_LoadCompleted(object sender, NavigationEventArgs e)
        {
            WebViewBrush b = new WebViewBrush();
            b.SourceName = "WebView1";
            b.Redraw();
            Rectangle1.Fill = b;
        }
...
```

19.4.5 ToolTip

Als kleine Hilfe für den Anwender stehen auch in WinRT-Apps die bekannten Tooltips zur Verfügung. Diese können auf reinem Text oder auch auf komplexeren Kombinationen von Controls (Grafiken, Texte, evtl. auch kurzes Video) basieren.

Zugewiesen werden die *ToolTip*s über den *ToolTipService* und dessen Eigenschaften:

- *ToolTip* (der gewünschte Text oder auch ein Container-Control mit beliebigen Elementen)
- *ToolTipPlacement* (wo soll der *ToolTip* erscheinen)
- *PlacementTarget* (an welchem Control soll der *ToolTip* ausgerichtet werden)

Ein kleines Beispiel zeigt die Möglichkeiten.

BEISPIEL 19.31: Verwendung von *ToolTips*

XAML

```
...
        <StackPanel Grid.Row="1" Margin="120">
           <StackPanel Orientation="Horizontal">
```

Ein Rechteck mit zugehörigem *ToolTip,* der am Mauskursor ausgerichtet wird:

```
              <Rectangle Height="100" Width="100" Fill="#FFFB0000"
                         ToolTipService.ToolTip="Ein kleiner ToolTip mit Text"
                         ToolTipService.Placement="Mouse"  />
```

Ein *ToolTip* mit Grafik und Textausgabe:

```
              <Rectangle Height="100" Width="100" Fill="#FF2EFB00"
                         ToolTipService.Placement="Bottom" >
                 <ToolTipService.ToolTip>
                    <StackPanel>
                       <Image Source="Images/bulbon.png"/>
                       <TextBlock>Hier kann noch mehr Text stehen ...</TextBlock>
                       <Button HorizontalAlignment="Center">Klick mich!</Button>
                    </StackPanel>
                 </ToolTipService.ToolTip>
              </Rectangle>
```

Achtung: Obige Schaltfläche ist nur als (schlechtes) Beispiel eingefügt, Sie haben keine Möglichkeit diese zu bedienen!

Ändern der *ToolTip*-Ausrichtung:

```
              <Rectangle Height="100" Width="100" Fill="#FF0072FB"
                         ToolTipService.ToolTip="Ich gehöre zum blauen Rechteck!"
                         ToolTipService.Placement="Right" />
```

Ein *ToolTip* mit Video (als schlechtes Beispiel, aber es geht auch):

```
              <Rectangle Height="100" Width="100" Fill="#FFF0F00D"
                 ToolTipService.Placement="Right">
                 <ToolTipService.ToolTip>
                    <StackPanel>
                       <MediaElement Width="300" Height="200" Source="video.mts"
                                     AutoPlay="True" />
                    </StackPanel>
                 </ToolTipService.ToolTip>
              </Rectangle>
           </StackPanel>
        </StackPanel>
...
```

BEISPIEL 19.31: Verwendung von *ToolTips*

Ergebnis

Laufzeitansicht:

19.5 Praxisbeispiele

19.5.1 Einen StringFormat-Konverter implementieren

WPF-Programmierer werden bereits eine einfache Möglichkeit vermisst haben, gebundene Felder mittels *StringFormat* an die lokalen Gegebenheiten anzupassen. Da die Eigenschaft *StringFormat* fehlt bleibt nichts anderes übrig, als sich mit einem eigenen Format-Konverter zu behelfen.

Quellcode Klasse StringFormatConverter

Erstellen Sie eine neue Klasse *StringFormatConverter* und implementieren Sie die beiden für einen Converter erforderlichen Methoden *Convert* und *ConvertBack*:

```
public class StringFormatConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        if (parameter == null) return value;
        return String.Format((String)parameter, value);
    }

    public object ConvertBack(object value, Type targetType, object parameter, string language)
    {
        return value;
    }
}
```

HINWEIS: Wie Sie sehen, nutzen wir den Parameter des Konverters zur Zeichenformatierung.

Verwendung des Konverters

Um den Konverter in Ihrer App nutzen zu können, müssen Sie diesen zunächst in den Ressourcen deklarieren:

```
...
<Page.Resources>
    <common:StringFormatConverter x:Key="StringFormatConverter" />
</Page.Resources>
...
```

Nachfolgend lässt sich der Konverter zuweisen und parametrieren:

Verwendung bei einem Datumswert:

```
<TextBlock Text="{Binding Datum, ElementName=pageRoot, ConverterParameter='{}{0:dd. MMMM yyyy}',
           Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
           Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Datum, ElementName=pageRoot, ConverterParameter='{}{0:dd.MM.yyyy}',
           Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
           Style="{StaticResource SubheaderTextStyle}"/>
```

Verwendung bei einem Gleitkommawert:

```
<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{}{0:C}',
           Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
           Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{}{0:E}',
           Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
           Style="{StaticResource SubheaderTextStyle}"/>
<TextBlock Text="{Binding Preis, ElementName=pageRoot, ConverterParameter='{}{0:G}',
           Converter={StaticResource StringFormatConverter}}" HorizontalAlignment="Left"
           Style="{StaticResource SubheaderTextStyle}"/>
```

Zuweisen der Beispielwerte im Page-Konstruktor:

```
...
    public sealed partial class BasicPage1 : StringFormatKonverter.Common.LayoutAwarePage
    {
        public DateTime Datum { get; set; }
        public Double Preis { get; set; }

        public BasicPage1()
        {
            this.InitializeComponent();
            Datum = System.DateTime.Now.AddMonths(5);
            Preis = 123456.78;
        }
...
```

Test

Die Ausgabe:

```
14. November 2012
14.11.2012
123.456,78 €
1,234568E+005
123456,78
```

19.5.2 Besonderheiten der TextBox kennen lernen

Für den von Windows Forms kommenden Umsteiger dürften sich bei Verwendung der *TextBox* einige Aha-Erlebnisse einstellen. Auf den ersten Blick nicht ersichtlich, bietet die *TextBox* in WinRT-Apps einen erweiterten Funktionsumfang, der auf die Touch-Unterstützung zielt.

Virtuelle Tastatur

Entwickeln und nutzen Sie Ihre Apps auf einem normalen Desktop-PC, werden Sie bei der *TextBox* keinen Unterschied zu den aus den .NET-Anwendungen bekannten TextBoxen erkennen. Ganz anders aber ist das Verhalten, wenn Sie ein Tablet mit Toucheingabe oder den Simulator verwenden. Hier wird beim Fokuserhalt eine zusätzliche virtuelle Tastatur eingeblendet, die überhaupt erst eine Texteingabe ermöglicht:

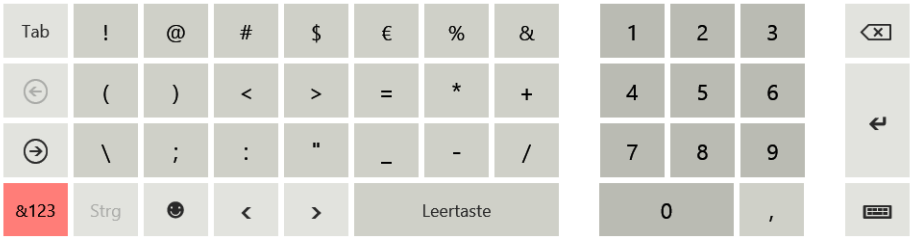
Je nach Einsatzzweck können Sie diese Tastatur auch anpassen. Nutzen Sie dafür die *InputScopeName*-Eigenschaft, die Sie allerdings nur recht umständlich setzen können:

```
<TextBox Text="Eingabetext">
    <TextBox.InputScope>
        <InputScope>
            <InputScope.Names>
                <InputScopeName NameValue="Number"/>
            </InputScope.Names>
        </InputScope>
    </TextBox.InputScope>
</TextBox>
```

Für deutsche Apps sind folgende Werte relevant:

- *Url*
- *EmailSmtpAddress*
- *Number*
- *TelephoneNumber*
- *Search*
- *AlphanumericHalfWidth*
- *AlphanumericFullWidth*

Die virtuelle Tastatur für Zahleneingaben:



Die virtuelle Tastatur für E-Mail-Adresseingaben:

HINWEIS: Das Setzen dieser Eigenschaft hat keinen Einfluss auf die zulässigen Werte in der *TextBox*, es wird lediglich die virtuelle Tastatur angepasst, um das Filtern der Tastatureingaben müssen Sie sich nach wie vor selbst kümmern.

Tasteneingaben validieren/filtern

Im Unterschied zu den guten alten TextBoxen in Windows Forms oder WPF stehen Ihnen in WinRT-Apps kaum Möglichkeiten zum Validieren und Filtern von Eingaben zur Verfügung. An dieser Stelle bleibt Ihnen nichts anderes übrig, als sich eine eigene Logik für das *TextBox*-Control zu programmieren.

Der wohl einfachste und naheliegendste Ansatzpunkt ist zunächst das Beschränken der per Tastatur einzugebenden Zeichen. Nutzen Sie dazu das *KeyDown*-Ereignis:

```
        private void TextBox_KeyDown_1(object sender, KeyRoutedEventArgs e)
        {
            e.Handled = (e.Key < Windows.System.VirtualKey.Number0 |
                         e.Key > Windows.System.VirtualKey.Number9);
        }
```

Diese Routine filtert alle Zeichen außerhalb des Bereichs 0... 9 heraus.

Ein Test wird auf den ersten Blick die Funktion bestätigen, doch probieren Sie mal einen Text per Zwischenablage in die *TextBox* einzufügen. Hier nützt uns obige Routine nichts, wir müssen zusätzliche Prüfungen im *TextChanged*-Ereignis durchführen:

Grundlage ist zunächst der Ursprungszustand der *TextBox* (dieser sollte den Regeln entsprechen). Diesen speichern wir im *GotFocus*-Ereignis ab:

```
...
        string txt1value = "";
...
        private void TextBox_GotFocus_1(object sender, RoutedEventArgs e)
        {
            txt1value = (sender as TextBox).Text;
        }
```

Ändert sich der Inhalt der *TextBox* (z.B. Einfügen der Zwischenablage), führen wir folgende Ereignisprozedur aus:

```
        private void TextBox_TextChanged_1(object sender, TextChangedEventArgs e)
        {
```

Eine Fehlerbehandlung für den Ernstfall:

```
            try
            {
```

Alle Zeichen außer 0...9 herausfiltern:

```
                Regex regex = new Regex("[^0-9]");
                string newvalue = regex.Replace((sender as TextBox).Text,  String.Empty);
```

Sollte kein Wert mehr vorhanden sein, setzen wir den Wert auf Null:

```
            if (newvalue == String.Empty) newvalue = "0";
```

Zur Sicherheit testen wir die Umwandlung in *Int32*:

```
            Int32 val = Convert.ToInt32(newvalue);
```

Hat alles geklappt, übernehmen wir den Wert:

```
            txt1value = newvalue;
            (sender as TextBox).Text = newvalue;
        }
        catch (Exception)
        {
```

Im Fehlerfall stellen wir den ursprünglichen Wert wieder her[1]:

```
            (sender as TextBox).Text = txt1value;
        }
    }
```

Damit beschränkt sich die Eingabe in der *TextBox* auf gültige *Int32*-Zahlen.

19.5.3 Daten in der GridView gruppieren

Ordnung ist das halbe Leben und so kommt schnell der Wunsch auf, lange Listen nicht nur optisch, sondern auch logisch zu sortieren bzw. zu gruppieren. Kein Problem, das *GridView*-Control besitzt bereits alle Voraussetzungen dafür, Sie müssen "nur noch" die Daten nach Ihren Wünschen gruppieren. Ganz nebenbei benötigen Sie noch eine *CollectionViewSource*, mit deren Hilfe wir die erforderliche Gruppierung vornehmen. Doch der Reihe nach …

Oberfläche

Erstellen Sie ein neues WinRT-Projekt, fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu und legen Sie deren Eigenschaft *IsSourceGrouped* mit *True* fest:

```
...
  <Page.Resources>
      <common:StringFormatConverter x:Key="StringFormatConverter" />
      <CollectionViewSource x:Name="CollectionViewSource1" IsSourceGrouped="True" />
   </Page.Resources>
...
```

Die *CollectionViewSource* ist quasi der Mittler zur internen *CollectionView* mit deren Hilfe die Daten gruppiert werden. Statt die Daten direkt an das *GridView*-Control zu binden (*ItemsSource*), schieben wir jetzt die *CollectionViewSource* dazwischen.

```
...
        <StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Vertical">
            <!-- Bindung an eine CollectionViewSource →
```

[1] Eine Undo-Methode ist leider nicht vorhanden.

Die Bindung an die *CollectionViewSource*:

```
<GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
        Background="AliceBlue" ScrollViewer.VerticalScrollMode="Enabled"
        ScrollViewer.VerticalScrollBarVisibility="Visible" Height="200"
        Width="800"  VerticalAlignment="Top" HorizontalAlignment="Left">
```

Für unser *GridView* haben wir zusätzlich die vertikalen Scrollbars aktiviert, so können die Gruppen später eine beliebige Höhe erreichen (die Gruppen werden vertikal angeordnet).

Das folgende *ItemTemplate* bestimmt das Aussehen des einzelnen Items (Rahmen, Inhalt, Abstand zum nächsten Item):

```
<GridView.ItemTemplate>
    <DataTemplate>
        <Border Background="#FFFFFBB1" Width="150" Height="45" Padding="5,0,5,0"
                BorderBrush="#FFF93131">
            <StackPanel>
```

Der Inhalt besteht aus zwei *TextBlock*-Controls mit den anzuzeigenden Daten[1]:

```
                <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"/>
                <TextBlock Text="{Binding Preis, ConverterParameter='{}{0:C}',
                   Converter={StaticResource StringFormatConverter}}"
                   HorizontalAlignment="Left"/>
            </StackPanel>
        </Border>
    </DataTemplate>
</GridView.ItemTemplate>
```

Mit dem *ItemsPanelTemplate* bestimmen Sie die spätere Anordnung und Gestaltung der einzelnen Gruppen. In diesem Fall verwenden wir ein *StackPanel,* die einzelnen Gruppen werden im vorliegenden Fall horizontal angeordnet:

```
<GridView.ItemsPanel>
    <ItemsPanelTemplate>
        <StackPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
</GridView.ItemsPanel>
```

Damit ist zunächst die Gruppe und der Gruppeninhalt bestimmt, was bleibt ist der Gruppenkopf, den Sie mit einem *HeaderTemplate* und einem *ItemsPanelTemplate* formatieren können:

```
<GridView.GroupStyle>
    <GroupStyle>
        <GroupStyle.HeaderTemplate>
            <DataTemplate>
```

Die Daten für den Inhalt des Gruppenkopfs stammen aus der *CollectionViewSource*, wir binden an das Feld *Key*, das beim Gruppieren automatisch erstellt wird:

```
                <TextBlock Text='{Binding Key}' Foreground="Gray" FontSize="25"
```

[1] Den *StringFormatConverter* kennen Sie bereits aus dem vorhergehenden Beispiel.

```
                                    Margin="5" />
                        </DataTemplate>
                    </GroupStyle.HeaderTemplate>
                    <GroupStyle.Panel>
                        <ItemsPanelTemplate>
                            <StackPanel  Margin="4" Background="Brown" />
                        </ItemsPanelTemplate>
                    </GroupStyle.Panel>
                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>
```

HINWEIS: Unter Bemerkungen finden Sie weitere alternative Gestaltungsmöglichkeiten

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 19.19 in das Projekt und passen Sie den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

```
...
    public sealed partial class BasicPage1 : GridViewGruppieren.Common.LayoutAwarePage
    {
        public BasicPage1()
        {
            this.InitializeComponent();
```

Aus unserer einfachen Collection von *Artikel*-Objekten erstellen wir mit folgender LINQ-Abfrage eine gruppierte Collection, die wir jetzt der *CollectionViewSource* (nicht dem *GridView*) zuweisen:

```
            var result = from art in Artikel.GetBeispielArtikel()
                         group art by art.Name.Substring(0,1)
                         into grp orderby grp.Key select grp;
```

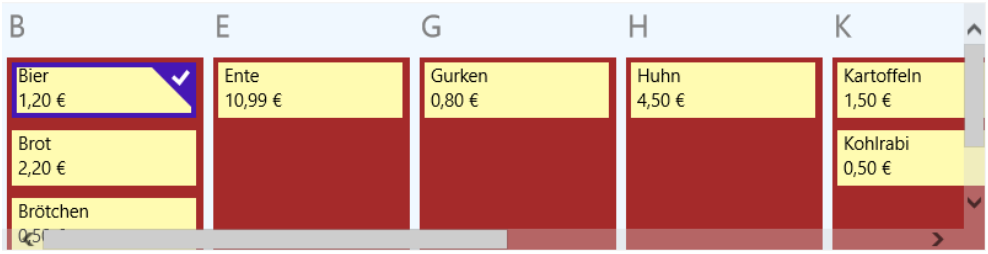
HINWEIS: Als Gruppierungsattribut verwenden wir den ersten Buchstaben des Artikelnamens. Sie könnten auch problemlos nach einem kompletten Feld oder auch nach unterschiedlichen Preisen gruppieren lassen.

```
            CollectionViewSource1.Source = result;
...
```

Damit ist das Beispiel fertig und wir können einen ersten Test wagen.

Test

Erwartungsgemäß erscheinen die Artikel jetzt in Gruppen, die nach dem Anfangsbuchstaben des Artikels gebildet wurden:



HINWEIS: Wir haben das *GridView* bewusst klein gehalten, so erscheinen auch die Scrollbars eher (nur wenn das Control den Fokus besitzt).

Bemerkung

Im vorliegenden Beispiel werden beliebig viele Gruppenmitglieder senkrecht unter dem Gruppenkopf angezeigt. Wer auf einen vertikalen Scrollbar verzichten will, kann die Gruppen auch mit Hilfe eines *VariableSizedWrapGrid* anordnen und die Anzahl der Zeilen beschränken. In diesem Fall werden die Gruppenmitglieder in horizontaler Richtung "umgebrochen", die folgende Gruppe verschiebt sich weiter nach links.

```
          <GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
                    Background="AliceBlue"  Height="160"  Width="800"  VerticalAlignment="Top"
                    HorizontalAlignment="Left" Margin="0,50,0,0">
...
              <GridView.GroupStyle>
                  <GroupStyle>
                      <GroupStyle.HeaderTemplate>
                          <DataTemplate>
```

Wir ändern auch gleich noch das Aussehen des Gruppenkopfes mittels *Border*:

```
                              <Border Width="50" Height="33" Background="Brown"
                                  CornerRadius="10,10,0,0" Margin="4,0,0,0" >
                                  <TextBlock Text='{Binding Key}' Foreground="White"
                                             FontSize="25" HorizontalAlignment="Center"
                                             VerticalAlignment="Center" />
                              </Border>
                          </DataTemplate>
                      </GroupStyle.HeaderTemplate>
                      <GroupStyle.Panel>
                          <ItemsPanelTemplate>
```

Hier bestimmen wir, wie viele Zeilen pro Gruppe angezeigt werden:

```
                            <VariableSizedWrapGrid MaximumRowsOrColumns="2"
                               Orientation="Vertical" Margin="4,0,4,0" Background="Brown" />
                          </ItemsPanelTemplate>
```

```
                    </GroupStyle.Panel>
                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>
```

Das Ergebnis unserer Bemühungen:

Wem eine horizontale Anordnung der Gruppen nicht gefällt kann auch dies ändern, wie es das folgende Beispiel zeigt:

```
        <GridView ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
              Background="AliceBlue" ScrollViewer.VerticalScrollMode="Enabled"
              ScrollViewer.VerticalScrollBarVisibility="Visible" Height="300"  Width="500"
              VerticalAlignment="Top" HorizontalAlignment="Left" Margin="0,50,0,0">
...
```

Die Gruppen vertikal anordnen:

```
            <GridView.ItemsPanel>
                <ItemsPanelTemplate>
                    <StackPanel Orientation="Vertical"/>
                </ItemsPanelTemplate>
            </GridView.ItemsPanel>
            <GridView.GroupStyle>
                <GroupStyle>
...
                    <GroupStyle.Panel>
                        <ItemsPanelTemplate>
```

Die Gruppeninhalte horizontal anordnen:

```
                            <StackPanel Margin="4" Background="Brown"
                                        Orientation="Horizontal" />
                        </ItemsPanelTemplate>
                    </GroupStyle.Panel>
                </GroupStyle>
            </GridView.GroupStyle>
        </GridView>
```

Die umformatierte *GridView* zeigt die nächste Abbildung.

Weitere Informationen zur *CollectionViewSource* finden Sie ab Seite 981 im Praxisbeispiel

- 19.5.5 Die CollectionViewSource verwenden.

19.5.4 Das SemanticZoom-Control verwenden

Umfangreiche Listendarstellungen im *GridView* haben den Nachteil, dass der Nutzer möglicherweise die Übersicht verliert. Eine erste Lösungsmöglichkeit hat das vorhergehende Praxisbeispiel gezeigt: die Gruppierung innerhalb der *GridView*. Wer jedoch schon einmal mit Touch-Eingabegeräten gearbeitet hat wird sich vielleicht erinnern, dass es auch entsprechende Gesten für das Verkleinern/Vergrößern (ZoomIn/ZoomOut) gibt. In der Übersichtsansicht kann der Nutzer dann zum Beispiel zu einer bestimmten Gruppe springen.

Die gleiche Funktionalität bietet auch das so genannte *SemanticZoom*-Control. Sie definieren zwei Zustände, um das Umschalten zwischen diesen per Gestensteuerung und um die Anzeige der Übergangsanimation kümmert sich das Control.

Die grundsätzliche XAML-Struktur:

```
<SemanticZoom>
    <ZoomedInView>
```

Hier folgt die Definition der detaillierten Ansicht:

```
        <GridView></GridView>
    </ZoomedInView>
    <ZoomedOutView>
```

Hier definieren Sie das Aussehen der Übersicht:

```
        <GridView></GridView>
    </ZoomedOutView>
</SemanticZoom>
```

Der Clou: Klicken Sie in der Übersicht auf einen Eintrag, springt die Detailansicht auf eben diese Gruppe, d h., es bietet sich eine schnelle Navigationsmöglichkeit innerhalb der **gruppierten** Collection. Wie Sie die Collection gruppiert anzeigen, haben wir bereits im vorhergehenden Praxisbeispiel 19.5.3 gezeigt.

HINWEIS: Wir verwenden als Datenquelle die Artikelliste aus dem Beispiel 19.19.

Oberflächen

Erstellen Sie ein neues WinRT-Projekt, fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu und legen Sie deren Eigenschaft *IsSourceGrouped* mit *True* fest:

```
...
   <Page.Resources>
       <common:StringFormatConverter x:Key="StringFormatConverter" />
       <CollectionViewSource x:Name="CollectionViewSource1" IsSourceGrouped="True" />
    </Page.Resources>
...
       <StackPanel Grid.Row="1" Margin="120,0,120,0" Orientation="Vertical">
```

Im *SemanticZoom*-Control realisieren wir zunächst den *ZoomedOutView*-Zustand:

```
          <SemanticZoom x:Name="semanticZoom" VerticalAlignment="Bottom"  >
              <SemanticZoom.ZoomedOutView>
```

Die *GridView* wird an die bereits definierte *CollectionViewSource* gebunden, achten Sie darauf, dass bei diesem Grid der *Path* auf *CollectionGroups* festgelegt ist (das ist in unserem Fall die Collection mit den Anfangsbuchstaben):

```
                  <GridView Name="GridView1" ItemsSource="{Binding Path=CollectionGroups,
                      Source={StaticResource CollectionViewSource1}}" IsSwipeEnabled="True" >
                      <GridView.ItemTemplate>
```

Das Aussehen bestimmen Sie wie gewohnt mit einem *DataTemplate*:

```
                          <DataTemplate>
                              <TextBlock Text="{Binding Group.Key}"
                                         FontFamily="Segoe UILight" FontSize="22"
                                         Foreground="Black" />
                          </DataTemplate>
                      </GridView.ItemTemplate>
```

Die Größe der einzelnen Items und deren Ausrichtung kann mit dem *ItemsPanelTemplate* festgelegt werden:

```
                      <GridView.ItemsPanel>
                          <ItemsPanelTemplate>
                              <WrapGrid ItemWidth="60" ItemHeight="60"
                                MaximumRowsOrColumns="1" VerticalChildrenAlignment="Center"/>
                          </ItemsPanelTemplate>
                      </GridView.ItemsPanel>
```

Über den *ItemContainerStyle* können Sie bequem das Aussehen (nicht den Inhalt) der einzelnen Items steuern:

```
                      <GridView.ItemContainerStyle>
                          <Style TargetType="GridViewItem">
                              <Setter Property="Margin" Value="4" />
                              <Setter Property="Padding" Value="10" />
                              <Setter Property="BorderBrush" Value="Gray" />
                              <Setter Property="BorderThickness" Value="1" />
```

```
                        <Setter Property="HorizontalContentAlignment" Value="Center" />
                        <Setter Property="VerticalContentAlignment" Value="Center" />
                    </Style>
                </GridView.ItemContainerStyle>
            </GridView>
        </SemanticZoom.ZoomedOutView>
```

Damit ist die Definition der *ZoomOutView* abgeschlossen, wir können uns der *ZoomInView* zuwenden:

```
        <SemanticZoom.ZoomedInView>
```

Die *GridView* wird so wie im vorhergehenden Praxisbeispiel konfiguriert, wir gehen an dieser Stelle nicht erneut darauf ein:

```
            <GridView ItemsSource="{Binding Source={StaticResource
                CollectionViewSource1}}" Background="AliceBlue"
                ScrollViewer.VerticalScrollMode="Enabled"
                ScrollViewer.VerticalScrollBarVisibility="Visible" Height="200"
                      VerticalAlignment="Top" HorizontalAlignment="Left">
                <GridView.ItemTemplate>
                    <DataTemplate>
                        <Border Background="#FFFFFBB1" Width="250" Height="45"
                            Padding="5,0,5,0" BorderBrush="#FFF93131">
                            <StackPanel>
                                <TextBlock Text="{Binding Name}"
                                      HorizontalAlignment="Left"/>
                                <TextBlock Text="{Binding Preis,
                                    ConverterParameter='{}{0:C}',
                                    Converter={StaticResource StringFormatConverter}}"
                                    HorizontalAlignment="Left"/>
                            </StackPanel>
                        </Border>
                    </DataTemplate>
                </GridView.ItemTemplate>
                <GridView.ItemsPanel>
                    <ItemsPanelTemplate>
                        <StackPanel Orientation="Horizontal"/>
                    </ItemsPanelTemplate>
                </GridView.ItemsPanel>
                <GridView.GroupStyle>
                    <GroupStyle>
                        <GroupStyle.HeaderTemplate>
                            <DataTemplate>
                                <TextBlock Text='{Binding Key}' Foreground="Gray"
                                FontSize="25" Margin="5" />
                            </DataTemplate>
                        </GroupStyle.HeaderTemplate>
                        <GroupStyle.Panel>
                            <ItemsPanelTemplate>
```

```
                                    <VariableSizedWrapGrid Orientation="Vertical"
                                      Height="400" />
                                </ItemsPanelTemplate>
                            </GroupStyle.Panel>
                        </GroupStyle>
                    </GridView.GroupStyle>
                </GridView>
            </SemanticZoom.ZoomedInView>
        </SemanticZoom>
        <ToggleSwitch IsOn="{Binding ElementName=semanticZoom, Path=IsZoomedInViewActive,
             Mode=TwoWay}">Detailansicht</ToggleSwitch>
```

Quellcode

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 19.19 in das Projekt und passen Sie den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

```
...
        public BasicPage1()
        {
            this.InitializeComponent();
            var result = from art in Artikel.GetBeispielArtikel() group art
                         by art.Name.Substring(0, 1) into grp orderby grp.Key select grp;
            CollectionViewSource1.Source = result;
        }
```

Test

Nach dem Start wird Ihnen zunächst die Detailansicht (*ZoomInView*) angezeigt. Besitzen Sie ein touchfähiges Gerät nutzen Sie die Geste zum Verkleinern (zwei Finger zusammenführen), um in die Übersichtsansicht (*ZoomOutView*) zu wechseln. Alternativ können Sie auch mit dem Mausrad zusammen mit der Strg-Taste diesen Effekt erreichen.

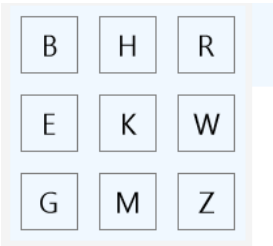
Die *ZoomInView*:

Die *ZoomOutView*:

Klicken Sie in der *ZoomOutView* auf den Buchstaben "Z", so sollte nach Rückkehr in die *ZoomInView* die Gruppe mit dem Anfangsbuchstaben "Z" in den sichtbaren Bereich gerückt sein.

HINWEIS: Alternativ kann die Ansicht auch über die Eigenschaft *IsZoomedInViewActive* gewechselt werden, wie wir es zum Beispiel mit dem *ToggleSwitch* realisiert haben.

Haben Sie beim *ItemsPanelTemplate* für die *ZoomOutView* den Wert von *MaximumRowsOrColumns* zum Beispiel auf 3 festgelegt, dürfte die *ZoomOutView* wie folgt aussehen:



19.5.5 Die CollectionViewSource verwenden

Wer sich durch die Praxisbeispiele dieses Kapitels gekämpft hat, wird bereits mehrfach in Kontakt mit der *CollectionViewSource* getreten sein. Dieses Control fungiert quasi als Mittler zur automatisch erstellten View (Satzzeigerverwaltung) bei einer Datenbindung. Da die aus WPF bekannte statische Methode *CollectionViewSource.GetDefaultView* für den Zugriff auf die View nicht zur Verfügung steht, stellt das Control den einfachsten Weg dar, um mit dem *CollectionView*-Objekt zu arbeiten (Eigenschaft *View*).

HINWEIS: Ein Sortieren und/oder Filtern ist bei der WinRT-Version der *CollectionViewSource* nicht möglich. Diese Funktionalität müssen Sie mittels LINQ realisieren. Damit beschränkt sich die Funktionalität in diesem Bereich auf das Gruppieren von Daten, wie es auch im Praxisbeispiel 19.5.3 ab Seite 972 beschrieben ist.

Was bleibt, ist die Verwaltung des Satzzeigers, die wir im folgenden kleinen Beispiel demonstrieren wollen.

Oberfläche

Erstellen Sie ein neues WinRT-Projekt und fügen Sie der *MainPage* zunächst eine *CollectionViewSource* hinzu:

```
...
    <Page.Resources>
        <CollectionViewSource x:Name="CollectionViewSource1" />
    </Page.Resources>
...
```

Unsere kleine App-Oberfläche enthält eine *ListBox*, die wir an die *CollectionViewSource* binden:

```
        <StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Horizontal">
            <ListBox ItemsSource="{Binding Source={StaticResource CollectionViewSource1}}"
              DisplayMemberPath="Name" Width="200" Height="400" VerticalAlignment="Top"/>
            <StackPanel Orientation="Vertical" Margin="40,0,0,0">
                <TextBlock>Navigation:</TextBlock>
```

Die folgenden Schaltflächen dienen als Navigationsschaltflächen (siehe auch Sourcecode):

```
                <StackPanel Orientation="Horizontal">
                    <Button Content="&#57600;" Click="Button_Click_2"
                            FontFamily="Segoe UI Symbol"/>
                    <Button Content="." Click="Button_Click_3" FontFamily="Segoe UI Symbol"/>
                    <Button Content="." Click="Button_Click_4" FontFamily="Segoe UI Symbol"/>
                    <Button Content="&#57601;" Click="Button_Click_5"
                            FontFamily="Segoe UI Symbol"/>
                    <Button Content="5" Click="Button_Click_6" FontFamily="Segoe UI Symbol"/>
                </StackPanel>
```

HINWEIS: Beachten Sie die Verwendung der Schriftart "Segoe UI Symbol", die mit Windows 8 eingeführt wurde. Diese erspart Ihnen in vielen Fällen die Verwendung von Grafiken.

Die aktuelle Position des Satzzeigers können wir über die Eigenschaft *CurrentPosition* der *View* abrufen:

```
                <TextBlock Margin="0,30,0,0">Position:</TextBlock>
                <TextBlock Text="{Binding Path=View.CurrentPosition,
                           ElementName=CollectionViewSource1}"
                           Style="{StaticResource SubheaderTextStyle}"/>
                <TextBlock Margin="0,30,0,0">Auswertung Event:</TextBlock>
```

Diesen Text setzen wir per *CurrentChanged*-Ereignis:

```
                <TextBlock Name="txt1" Text="" Style="{StaticResource SubheaderTextStyle}"/>
            </StackPanel>
        </StackPanel>
...
```

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 19.19 in das Projekt und passen Sie den Namespace an. Nachfolgend können wir uns um den Konstruktor der Seite kümmern:

```
...
        public BasicPage1()
        {
            this.InitializeComponent();
```

Daten abrufen und der *CollectionViewSource* zuweisen:

```
            CollectionViewSource1.Source = Artikel.GetBeispielArtikel();
```

Wir nutzen die beiden Ereignisse *CurrentChanged* (nach dem Satzzeigerwechsel) und *CurrentChanging* (vor dem Satzzeigerwechsel):

```
            CollectionViewSource1.View.CurrentChanged += View_CurrentChanged;
            CollectionViewSource1.View.CurrentChanging += View_CurrentChanging;
        }
```

Wenn der aktuelle Artikel-Datensatz "Ente" ist, soll sich die Satzzeigerposition nicht mehr verändern lassen (die Änderung könnten Sie von weiteren Bedingungen abhängig machen):

```
        void View_CurrentChanging(object sender, CurrentChangingEventArgs e)
        {
            if ((CollectionViewSource1.View.CurrentItem as Artikel).Name == "Ente")
            {
                if (e.IsCancelable)
                    e.Cancel = true;
            }
        }
```

Hat sich die Satzzeigerposition erfolgreich geändert, zeigen wir den Namen des aktuell gewählten Artikels an:

```
        void View_CurrentChanged(object sender, object e)
        {
            txt1.Text = (CollectionViewSource1.View.CurrentItem as Artikel).Name;
        }
```

Wechsel zum ersten Datensatz:

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            CollectionViewSource1.View.MoveCurrentToFirst();
        }
```

Wechsel zum vorhergehenden Datensatz:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            CollectionViewSource1.View.MoveCurrentToPrevious();
```

```
            if (CollectionViewSource1.View.IsCurrentBeforeFirst)
                CollectionViewSource1.View.MoveCurrentToFirst();
        }
```

Wechsel zum nächsten Datensatz:

```
        private void Button_Click_4(object sender, RoutedEventArgs e)
        {
            CollectionViewSource1.View.MoveCurrentToNext();
            if (CollectionViewSource1.View.IsCurrentAfterLast)
                CollectionViewSource1.View.MoveCurrentToLast();
        }
```

Wechsel zum letzten Datensatz:

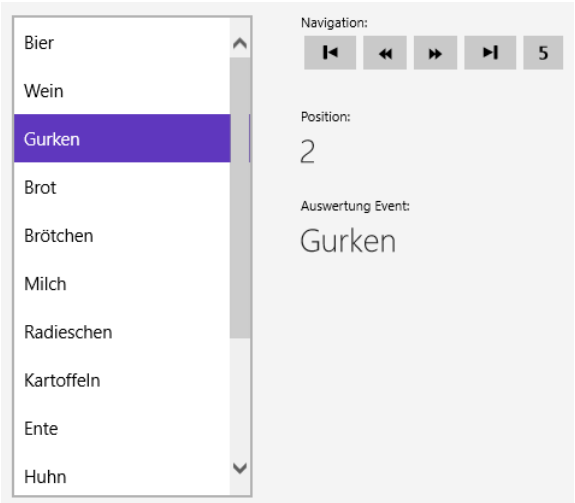
```
        private void Button_Click_5(object sender, RoutedEventArgs e)
        {
            CollectionViewSource1.View.MoveCurrentToLast();
        }
```

Sprung zum fünften Datensatz:

```
        private void Button_Click_6(object sender, RoutedEventArgs e)
        {
            CollectionViewSource1.View.MoveCurrentToPosition(5);
        }
...
```

Test

Nach dem Start der App können Sie mittels *ListBox* oder mit den Navigationstasten zwischen den Datensätzen hin- und herspringen. Haben Sie einmal den Artikel "Ente" ausgewählt, ist eine Änderung der Satzzeigerposition nicht mehr möglich.



19.5.6 Zusammenspiel ListBox/AppBar

Es wird immer wieder propagiert: Windows 8-Oberflächen sollen standardmäßig nur die notwendigsten Informationen anzeigen, weitere Optionen, Befehle sollen auf den AppBars am oberen oder unteren Bildschirmrand eingeblendet werden, wenn dies erforderlich ist. Leicht gesagt, doch wie sieht dies in der Praxis aus?

Unser Beispiel zeigt Ihnen wie Sie ein AppBar einblenden, wenn Einträge in einer *ListBox* (dies könnte auch jedes andere Listen-Control sein) ausgewählt werden bzw. wie Sie die *AppBar* ausblenden, wenn kein Eintrag selektiert ist.

Oberfläche

Erstellen Sie ein neues WinRT-Projekt und fügen Sie der *MainPage* eine *CollectionViewSource* hinzu:

```
...
    <Page.Resources>
        <CollectionViewSource x:Name="CollectionViewSource1" />
    </Page.Resources>
```

Wir erstellen zunächst den *AppBar* für den unteren Bildschirmrand:

```
    <Page.BottomAppBar>
        <AppBar x:Name="BottomAppBar1" Padding="10,0,10,0">
```

Ein Grid hilft uns bei der Ausrichtung (die Schaltflächen sollen wegen der Erreichbarkeit mit den Fingern möglichst am linken und rechten Rand liegen):

```
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="50*"/>
                    <ColumnDefinition Width="50*"/>
                </Grid.ColumnDefinitions>
```

Diese Schaltflächen werden am linken Rand ausgerichtet:

```
                <StackPanel Orientation="Horizontal" Grid.Column="0" HorizontalAlignment="Left">
                    <Button x:Name="Edit" Style="{StaticResource EditAppBarButtonStyle}"
                            Click="Edit_Click_1"/>
                    <Button x:Name="Delete" Style="{StaticResource DeleteAppBarButtonStyle}"
                            Click="Delete_Click_1" />
                </StackPanel>
```

Diese Schaltfläche wird rechts unten angezeigt:

```
                <StackPanel Orientation="Horizontal" Grid.Column="1"
                            HorizontalAlignment="Right">
                    <Button x:Name="Help" Style="{StaticResource HelpAppBarButtonStyle}"/>
                </StackPanel>
            </Grid>
        </AppBar>
    </Page.BottomAppBar>
```

```
...
```

Als Beispiel für eine Datenanzeige dient uns eine *ListBox*, welche wir an die *CollectionViewSource* binden:

```
      <StackPanel Grid.Row="1" Margin="120,0,20,0" Orientation="Horizontal">
          <ListBox Name="ListBox1" ItemsSource="{Binding Source={StaticResource
             CollectionViewSource1}}" DisplayMemberPath="Name" Width="200" Height="400"
             VerticalAlignment="Top"  SelectionMode="Multiple"
          SelectionChanged="ListView_SelectionChanged" />
       </StackPanel>
...
```

HINWEIS: Mit *SelectionChanged* reagieren wir zur Laufzeit auf Änderungen der Auswahl in der *ListBox*.

Quelltext

Kopieren Sie zunächst die Klasse *Artikel* aus dem Beispiel 19.19 in das Projekt und passen Sie den Namespace an. Nachfolgend kümmern wir uns um den Konstruktor der Seite:

```
...
    public sealed partial class BasicPage1 : Bsp_AppBar.Common.LayoutAwarePage
    {
        public BasicPage1()
        {
            this.InitializeComponent();
```

Daten laden:

```
            CollectionViewSource1.Source = Artikel.GetBeispielArtikel();
        }
```

Ändert sich die Auswahl in der *ListBox* (mehr als ein ausgewählter Item), blenden wir die *AppBar* ein (*IsOpen*) und sorgen dafür, dass diese auch geöffnet bleibt (*IsSticky*):

```
        private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
        {
            if (this.ListBox1.SelectedItems.Count > 0)
            {
                this.BottomAppBar1.IsSticky = true;
                this.BottomAppBar1.IsOpen = true;
            }
```

Keine Auswahl, keine Anzeige:

```
            else
            {
                this.BottomAppBar1.IsOpen = false;
                this.BottomAppBar1.IsSticky = false;
            }
        }
```

Klickt der Nutzer auf die *Delete*-Schaltfläche, zeigen wir eine Sicherheitsabfrage an:

```
        async private void Delete_Click_1(object sender, RoutedEventArgs e)
        {
            string msg = "Datensatz wirklich löschen?";
            if (ListBox1.SelectedItems.Count > 1)
                msg = "Datensätze wirklich löschen?";
```

MessageDialog erzeugen:

```
            var messageDialog = new MessageDialog(msg, "Sicherheitsabfrage");
```

Zwei Schaltflächen einblenden:

```
            messageDialog.Commands.Add(new UICommand("Ja",
                new UICommandInvokedHandler(this.CommandInvokedHandler),true));
            messageDialog.Commands.Add(new UICommand("Nein",
                new UICommandInvokedHandler(this.CommandInvokedHandler), false));
            messageDialog.DefaultCommandIndex = 1;
            messageDialog.CancelCommandIndex = 1;
```

Anzeige des Dialogs:

```
            await messageDialog.ShowAsync();
        }
```

Das Ergebnis des Dialogs können wir in folgendem Ereignis auswerten:

```
        private void CommandInvokedHandler(IUICommand command)
        {
```

Wurde die Ja-Schaltfläche gedrückt, löschen wir alle markierten Einträge:

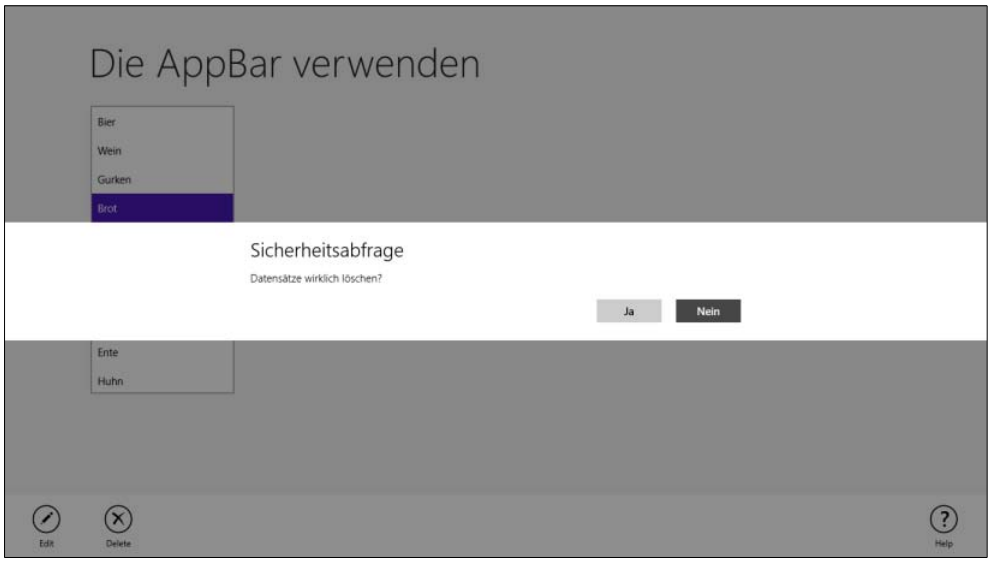
```
            if ((bool)command.Id)
            {
                while (ListBox1.SelectedItems.Count > 0)
                {
                    (CollectionViewSource1.Source as
                           ObservableCollection<Artikel>).Remove(
                               ListBox1.SelectedItems[0] as Artikel);
                }
            }
        }
```

Die Reaktion auf die *Edit*-Schaltfläche ist nur eine Statusmitteilung:

```
        async private void Edit_Click_1(object sender, RoutedEventArgs e)
        {
            var messageDialog = new MessageDialog("Funktion nicht implementiert!", "Hinweis");
            await messageDialog.ShowAsync();
        }
    }
...
```

Test

Öffnen Sie die App, markieren Sie einige Einträge in der *ListBox* und versuchen Sie diese per *AppBar* zu löschen:



19.5.7 Musikwiedergabe im Hintergrund realisieren

Sicher ist Ihnen auch schon aufgefallen, dass eine App, sobald sie nicht mehr den Fokus besitzt, "eingeschläfert" wird. So schön dieses auch für die Systemressourcen ist, so nachteilig wirkt es sich auf einige Verhaltensweisen von Programmen aus. Wollten Sie in der Vergangenheit beispielsweise Musik hören, so hatten Sie einen Player geöffnet, dort einen Titel ausgewählt und dann meist ein anderes Programm Ihrer Wahl gestartet, um damit zu arbeiten.

Diese Funktionalität steht Ihnen mit WinRT-Apps zunächst nicht zur Verfügung, ein Programm das in den Hintergrund gerät wird zunächst einmal deaktiviert. Damit endet auch die Musikwiedergabe.

Wie es trotzdem funktioniert zeigt das folgende Beispiel.

Oberfläche

Uns genügen einige Steuerungstasten (Play, Stop, Pause) sowie eine Schaltfläche für das Laden des Titels (auf eine umständliche Dateiauswahl verzichten wir an dieser Stelle, mehr dazu im Beispiel 17.3 ab Seite 876). Zusätzlich brauchen wir natürlich noch ein *MediaElement* für die Wiedergabe:

```
...
       <StackPanel Grid.Row="1" Margin="120,0,120,0">
           <MediaElement Width="1024" Height="720" Margin="0,0,0,30" Name="MediaElement1"
                         AutoPlay="False" AudioCategory="BackgroundCapableMedia"
                         CurrentStateChanged="MediaElement1_CurrentStateChanged" />
           <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
```

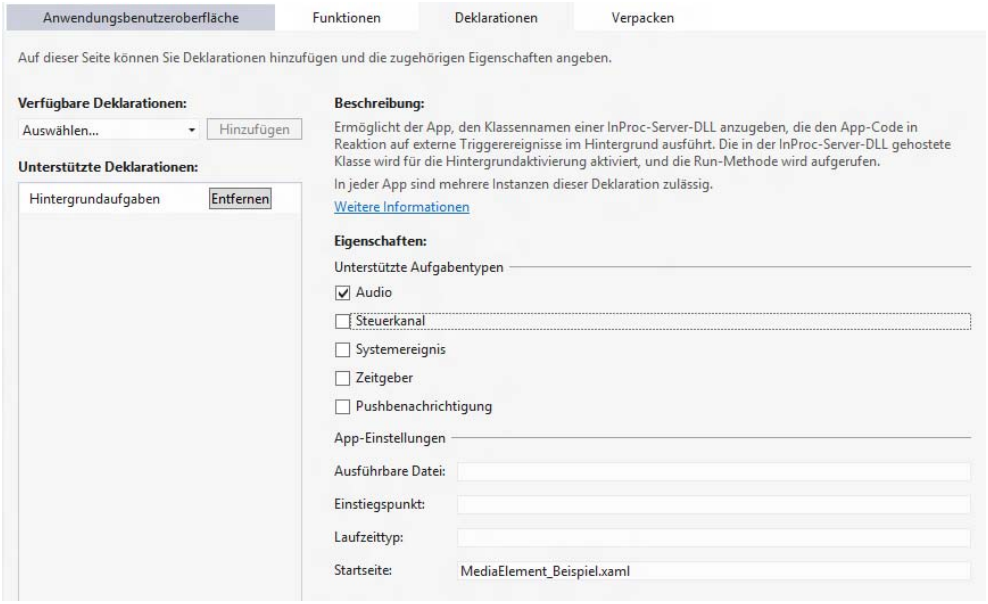
```
            <Button Click="Button_Click_1" Width="100">Start</Button>
            <Button Click="Button_Click_3" Width="100">Stop</Button>
            <Button Click="Button_Click_2" Width="100">Pause</Button>
            <Button Click="Button_Click_5" Width="100">Load MP3</Button>
        </StackPanel>
    </StackPanel>
...
```

HINWEIS: Wie Sie sehen, konfigurieren wir bereits an dieser Stelle das *MediaElement* für eine Hintergrundausgabe der Musik. Dies ist eine unbedingte Voraussetzung, um das Standardverhalten (Suspending) zu ändern.

Über das *CurrentStateChanged*-Ereignis werten wir zur Laufzeit den aktuellen Status der Wiedergabe aus und steuern damit die zentralen Steuerungstasten der Windows 8-Oberfläche.

Erstellen Sie im Projektmappen-Explorer weiterhin einen Unterordner *Musik,* in den Sie eine oder mehrere MP3-Dateien kopieren. Diese werden wir zur Laufzeit abspielen.

Neben dieser Änderung, müssen Sie auch eine "Hintergrundaufgabe" definieren, um WinRT zu einem geänderten Verhalten beim Umschalten zwischen den Apps zu veranlassen. Öffnen Sie dazu die Datei *Package.appxmanifest* und erstellen in der Rubrik *Deklarationen* eine neue "Hintergrundaufgabe", die den Aufgabentyp "Audio" unterstützt:



Nachfolgend widmen wir uns noch der Programmlogik.

Quelltext

Neben den bereits erfolgten Änderungen gegenüber einer "normalen" Soundausgabe muss Ihr Programm sich noch in das Handling der Medienwiedergabe "einklinken". Dazu sollte Ihre App die folgenden zentralen Ereignisse unterstützen:

```
...
using Windows.Media;
...
     private void pageRoot_Loaded_1(object sender, RoutedEventArgs e)
        {
            MediaControl.PlayPressed += MediaControl_PlayPressed;
            MediaControl.PausePressed += MediaControl_PausePressed;
            MediaControl.PlayPauseTogglePressed += MediaControl_PlayPauseTogglePressed;
            MediaControl.StopPressed += MediaControl_StopPressed;
        }
```

Wir reagieren auf die Hardware-Media-Tasten, müssen dabei aber beachten, dass diese aus einem anderen Context heraus aufgerufen werden:

```
        async void MediaControl_StopPressed(object sender, object e)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                MediaElement1.Stop();
            });
        }
```

Bei Wechsel zwischen Pause und Play werten wir den Status der entsprechenden Media-Taste aus (diesen setzen wir im Ereignis *CurrentStateChanged*):

```
        async void MediaControl_PlayPauseTogglePressed(object sender, object e)
        {
            if (MediaControl.IsPlaying)
            {
                await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
                {
                    MediaElement1.Pause();
                });
            }
            else
            {
                await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
                {
                    MediaElement1.Play();
                });
            }
        }
```

Reaktion auf die Einzeltasten:

```
        async void MediaControl_PausePressed(object sender, object e)
```

```
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                MediaElement1.Pause();
            });
        }

        async void MediaControl_PlayPressed(object sender, object e)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                MediaElement1.Play();
            });
        }
```

Ändert unser *MediaElement* den Status, setzen wir die Tasten für die zentrale Medienwiedergabe entsprechend:

```
        private void MediaElement1_CurrentStateChanged(object sender, RoutedEventArgs e)
        {
            if (MediaElement1.CurrentState == MediaElementState.Playing)
            {
                MediaControl.IsPlaying = true;
            }
            else
            {
                MediaControl.IsPlaying = false;
            }
        }
```

Ach ja, fast hätten wir vor lauter "Nebenaufgaben" das eigentliche Laden der MP3-Datei vergessen:

```
      private async void Button_Click_5(object sender, RoutedEventArgs e)
        {
            StorageFolder folder = await
                  Package.Current.InstalledLocation.GetFolderAsync("musik");
            StorageFile file = await folder.GetFileAsync("sound0.mp3");
            var stream = await file.OpenAsync(FileAccessMode.Read);
            MediaElement1.SetSource(stream, file.ContentType);
            MediaElement1.IsLooping = true;
        }
```

HINWEIS: Wir setzen die Soundwiedergabe auf "Endlos", so haben wir später genügend Zeit zum Testen.

Die Wiedergabetasten in unserer App:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

```
            MediaElement1.Play();
        }

        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            MediaElement1.Stop();
        }

        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            MediaElement1.Pause();
        }
```

Test

Starten Sie die App und laden Sie die MP3-Datei. Klicken Sie nachfolgend auf die Play-Taste der App. Spielt die Musik, wechseln Sie nun zu einer anderen App. Die Musik sollte weiterspielen. Klicken Sie jetzt auf die eventuell vorhandenen Media-Tasten Ihrer Tastatur um folgendes Popup-Fenster anzuzeigen:

Die hier dargestellte Pause-Taste korrespondiert mit dem Ereignis *PlayPauseTogglePressed* unserer App. Im Gegenzug setzt unsere App den Status dieser Taste per *CurrentStateChanged*-Ereignis.

Bemerkung

Die Hintergrundwiedergabe funktioniert also ganz gut, eines sollten Sie jedoch beachten:

HINWEIS: Besteht Ihre App aus mehreren Seiten und wird zwischen diesen Seiten gewechselt, nützt Ihnen die obige Hintergrundwiedergabe nichts, da die aktuelle Instanz des *MediaElement*-Controls mit dem Schließen der Seite zerstört wird. In diesem Fall müssen Sie eine zentrale Seite erstellen, in die Sie das *MediaElement* einfügen. Die Detailseiten wechseln dann nur noch innerhalb eines *Frame*-Controls in der zentralen Seite.

Kapitel 20

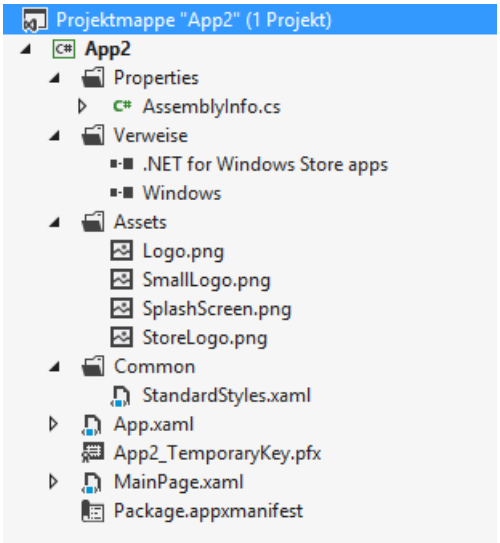
Apps im Detail

In diesem Kapitel wollen wir die WinRT-Apps näher beleuchten. Dazu nehmen wir zunächst eine App "auseinander", sehen uns den Lebenszyklus an und tauchen ein in die Möglichkeiten des Daten- und Dateizugriffs unter den Bedingungen von WinRT.

20.1 Ein Windows Store App-Projekt im Detail

Sicher kann es nicht schaden, wenn wir uns zunächst etwas intensiver mit dem Grundgerüst einer WinRT-App beschäftigen, zumal der administrative Aufwand im Laufe der Jahre und mit jeder neuen Zielumgebung immer komplexer geworden ist. WinRT-Apps machen da sicher keine Ausnahme, ein schnelles "hello world!"-Programm werden Sie kaum in wenigen Sekunden programmieren – vom Vertrieb an den Endkunden ganz zu schweigen.

Der Blick in ein neues, "leeres" WinRT-Projekt erzeugt zunächst nichts als Ratlosigkeit, es sei denn, Sie haben sich vorher schon intensiv mit WPF oder Silverlight beschäftigt.



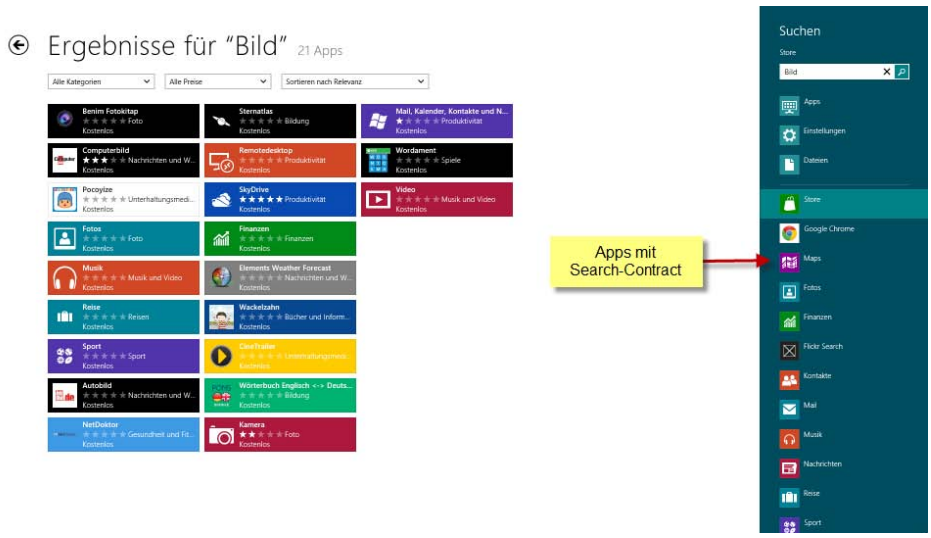
20.1.1 Contracts und Extensions

Bevor wir uns mit den einzelnen Dateien des Projekts beschäftigen können, müssen wir zunächst etwas weiter ausholen und uns mit einem der Grundkonzepte von WinRT-Apps vertraut machen.

Im Zusammenhang mit WinRT-Apps fallen häufig die Begriffe "Contract" und "Extension". Im Weiteren wollen wir es auch bei diesen engl. Begriffen belassen, um Fehlinterpretationen bei Verwendung der Hilfe etc. zu vermeiden. Doch worum handelt es sich eigentlich?

- **Contract**
  Hierbei geht es um einen "Vertrag" über eine Interaktion zwischen verschiedenen Apps bzw. dem WinRT-System. Dies kann zum Beispiel die Auswahl einer Datei (eine andere App möchte eine Datei laden und nutzt dazu Ihre App), eine Suchanforderung (durch das System) oder das Aktivieren eines Laufwerks (USB-Stick, SD-Card) sein. Der nötige Contract wird in der Manifestdatei *Package.appxmanifest* deklariert und konfiguriert (siehe Abschnitt 20.1.5, Seite 998), so kann WinRT später die gewünschten Aufgaben anfordern (WinRT ist hier nur der Vermittler zwischen den einzelnen Vertragspartnern). Ein entsprechender Callback in der Datei *App.xaml.cs* ermöglicht Ihrem Programm die Reaktion auf die externen Anforderungen durch das System oder durch eine andere App[1].

  Die folgende Abbildung zeigt als Beispiel die Suchfunktion der WinRT-Oberfläche. Per Search-Contract haben sich diverse Apps in das Windows-Suchsystem "eingeklinkt", in diesem Fall nutzen wir die zentrale Suche (das Flyout am rechten Bildrand), um im Windows-Store nach einer App zu suchen:



[1] Die grundlegendste Form eines Contracts ist der Launch-Contract, dieser ermöglicht überhaupt erst den Start der App. Den entsprechenden Callback *OnLaunched* finden Sie bereits vordefiniert in der Datei *App.xaml.cs* vor. Ohne diesen Contract startet keine WinRT-App.

Die Suchfunktion stellt also lediglich die Auswahl der Such-App und eine Eingabemöglichkeit zur Verfügung, die eigentliche Suche und die Anzeige der Suchergebnisse ist Aufgabe der App.

HINWEIS: Wie auch Sie Ihre App für die Suche registrieren, zeigt Ihnen das Praxisbeispiel 20.4.1 "Unterstützung für den Search-Contract bieten", ab Seite 1030.

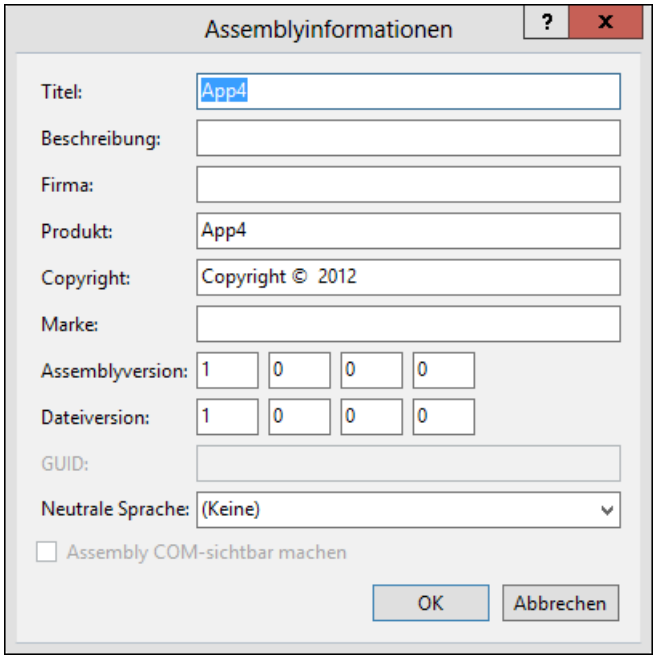
- **Extension**
  Mit Hilfe von Extensions werden Windowsfunktionen erweitert bzw. angepasst, um diese in der eigenen App oder eventuell auch in anderen Apps zu nutzen. Dies kann zum Beispiel der Zugriff auf die Kamera-Einstellungen, die Druckereinstellungen oder auch die Auto-Play-Funktion sein. Sie brauchen also nicht jedes Mal "das Rad komplett neu zu erfinden", sondern können sich quasi wie in einem Baukasten bei vorhandenen Grundgerüsten bedienen.

HINWEIS: Ein Beispiel für die Unterstützung der AutoPlay-Funktion finden Sie im Praxisbeispiel 20.4.2 ab Seite 1037.

Nach diesem kurzen theoretischen Ausflug können wir uns nun endlich den einzelnen Dateien des WinRT-Projekts widmen.

20.1.2 AssemblyInfo.cs

Diese Datei im Unterverzeichnis *Properties* speichert alle Informationen, die Sie auch über den Dialog *Projekt|Einstellungen|Anwendung|Assemblyinformationen* vornehmen können:



Die Daten werden intern als Attribute gespeichert und können wie folgt im laufenden Programm abgefragt werden:

BEISPIEL 20.1: Abfrage von Assemblyinformationen

C#

```
...
using System.Reflection;
...
   public BasicPage1()
   {
```

Die einfache Variante funktioniert nur für einige Attribute (z.B. Versionsinformation):

```
      txt1.Text = typeof(BasicPage1).GetTypeInfo().Assembly.GetName().Version.ToString();
```

Mit der folgenden Variante können Sie alle gewünschten Attribute abrufen (jeweils eines):

```
      var txt = (AssemblyDescriptionAttribute)
                typeof(BasicPage1).GetTypeInfo().Assembly.GetCustomAttribute(
                typeof(AssemblyDescriptionAttribute));
      txt2.Text = txt.Description;
   }
...
```

HINWEIS: Wie Sie sehen, ist der Zugriff per Reflection auf die laufende Assembly in WinRT-Apps neu organisiert worden. Die alte Version mit *System.Reflection.Assembly.GetExecutingAssembly()* funktioniert **nicht** mehr!

20.1.3 Verweise

Standardmäßig verweist Ihre WinRT-App "lediglich" auf

- *.NET for Windows Store apps*
- und *Windows*

Während es sich beim ersten Verweis um das .NET-Core-Profile mit den entsprechenden Assemblies unter

```
C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5
```

handelt, ist *Windows* ein Verweis auf die Metadatendatei:

```
C:\Program Files (x86)\Windows Kits\8.0\References\CommonConfiguration\Neutral\Windows.winmd
```

und damit auf alle *Windows.*-Namespaces, die Sie in Ihrer App benötigen.

Die Portable Assemblies finden Sie, wenn erforderlich, unter:

```
C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETPortable\v4.5
```

20.1.4 App.xaml und App.xaml.cs

Hierbei handelt es sich um den Ausgangs- und den Endpunkt für jede WinRT-App. So ist in der Datei *App.xaml.cs* mit der Klasse *App* (abgeleitet von *Application*)

- zum einen der Einstiegspunkt in Ihre App zu finden (*OnLaunched*),
- zum anderen reagieren Sie hier mit dem Ereignis *OnSuspending* auf das Pausieren bzw. das stille Ende Ihrer App.

HINWEIS: Erste Details zum Start der App hatten wir bereits im Abschnitt 18.1.2 "Die Page, der Frame und das Window" ab Seite 901 beschrieben. Mehr zu den einzelnen Methoden und Ereignissen dieser Klasse finden Sie im Abschnitt 20.2 "Der Lebenszyklus einer WinRT-App" ab Seite 1007.

Nutzen Sie die Klasse *App* ebenfalls für globale Variablen/Objekte und die Unterstützung diverser Contracts (Suche, Filepicker etc.), wir kommen später noch darauf zurück.

HINWEIS: Beachten Sie, dass das Initialisieren eigener Objektinstanzen (z.B. im Konstruktor dieser Klasse) nicht zu viel Zeit in Anspruch nimmt, andernfalls wird die App von WinRT gnadenlos aus dem Speicher geworfen.

Neben der Code-Datei sollten Sie auch die XAML-Datei nicht vernachlässigen, in *App.xaml* können Sie zentrale Ressourcen für Ihre App bereitstellen.

BEISPIEL 20.2: Inhalt der Datei *App.xaml* mit zusätzlich definierter Ressource

XAML

```
...
 <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Common/StandardStyles.xaml"/>
```

Hier können Sie weitere zentrale Ressourcen definieren:

```
                <ResourceDictionary>
                    <local:KundenDataSource x:Key="kundenDataSource"/>
                </ResourceDictionary>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
...
```

HINWEIS: Ob es allerdings Sinn macht, wie in obigem Beispiel zentrale Instanzen in der *App.xaml* zu definieren, ist fraglich, da können Sie besser gleich die *App*-Klasse um eine Eigenschaft erweitern und haben Ihren Code zentral zusammengefasst.

Ganz nebenbei, der Zugriff per Code auf obige Ressource ist zum Beispiel per

```
KundenDataSource kundenDataSource = (KundenDataSource)App.Current.Resources["kundenDataSource"];
```

möglich. Da ist ein schlankes

```
App.KundenDataSource
```

wohl wesentlich einfacher.

20.1.5 Package.appxmanifest

Hierbei handelt es sich um das Manifest Ihrer WinRT-App. Dieses dient dem WinRT-System als Informationsquelle über die Funktionalität Ihrer App.

HINWEIS: Bevor Sie verzweifeln: Es handelt sich zwar um eine XML-Datei, Sie können jedoch einen recht komfortablen Editor für das Konfigurieren aller Einstellungen nutzen.

Der Editor ist in folgende vier Seiten eingeteilt:

- Anwendungsbenutzeroberfläche
- Funktionen
- Deklarationen
- Verpacken

Anwendungsbenutzeroberfläche

Diese Seite dürfte wohl für die wenigsten Fragen sorgen. So legen Sie zunächst den Anzeigenamen (dieser erscheint in der Übersicht aller Apps) und den Einstiegspunkt in Ihre App fest (Letzteren werden Sie wohl nie ändern).

Die Angaben zu den unterstützten Drehungen sind optional und haben keine erkennbare Auswirkung auf das Verhalten der App. Weiterhin finden Sie hier die Optionen für die Beschriftung (der Kurzname ist der Text auf der Kachel), das Logo (Symbol) in verschiedenen Formaten, sowie die Schrift- und die Hintergrundfarbe Ihrer App auf dem Windows 8-Desktop.

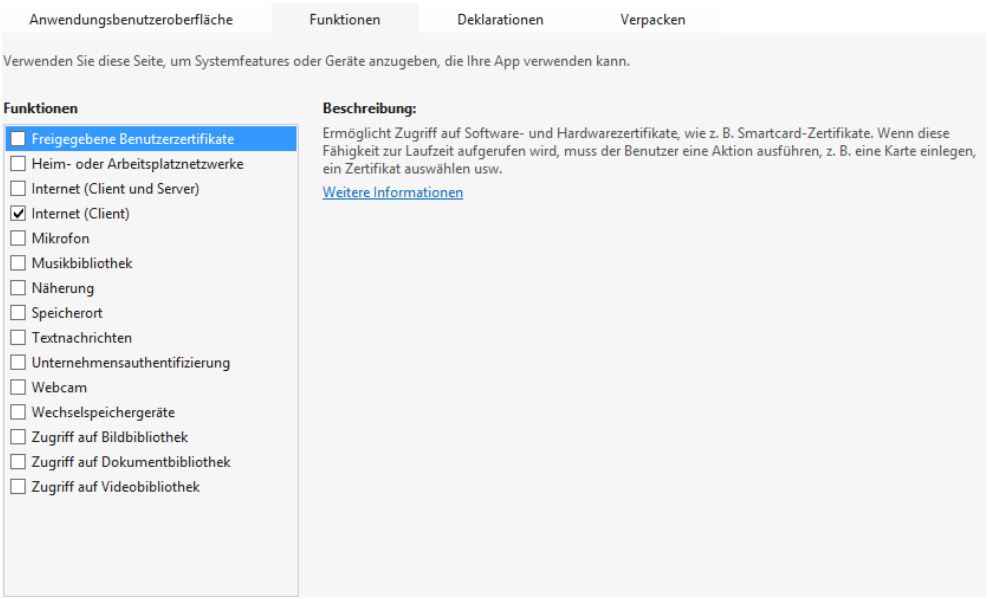
Optional können hier auch Symbole für Benachrichtigungen zugewiesen werden, wenn Ihre App diese Funktionalität unterstützt.

Last but not least, lassen sich auch das Logo sowie die Hintergrundfarbe des Splash Screens (Begrüßungsbildschirm) zuweisen.

Funktionen

Melden Sie auf dieser Seite an, welche Funktionen bzw. Fähigkeiten Ihre App unterstützt bzw. welche Funktionen Ihre App gern nutzen würde.

HINWEIS: Die letzte Aussage sollten Sie wörtlich nehmen. Auch wenn Sie zum Beispiel ein Häkchen bei Webcam setzen heißt das noch lange nicht, dass Ihre App auch darauf zugreifen kann. Einige Aktionen müssen beim ersten Start der App durch den Endanwender autorisiert werden.



Die folgende Tabelle gibt eine Übersicht zur Bedeutung der einzelnen Funktionen:

Funktion	Beschreibung
Freigegebene Benutzerzertifikate	Zugriff auf Hard- und Softwarezertifikate
Heim- und Arbeitsplatznetzwerk	Ein- und ausgehender Zugriff auf vertrauenswürdige Orte und Netzwerke
Internet (Client und Server)	Die App kann Daten aus dem Internet empfangen und gleichzeitig auf Anforderung auch Daten in das Internet senden.
Internet (Client)	Die App kann Daten aus dem Internet (Feeds etc.) abrufen.
Mikrofon/Webcam	Verwendung von Mikrofon und/oder Webcam anfordern. Beachten Sie, dass meist beide Optionen gesetzt werden müssen, wenn sich z.B. das Mikrofon in der Webcam befindet.
Zugriff auf Musikbibliothek/ Bildbibliothek/Dokumentenbibliothek/Videobibliothek	Schreib-/Lesezugriff auf die entsprechende Bibliothek freigeben
Näherung	Unterstützung für die neue Near-Field Communikation (NFC), die in Smartphones und Tablets zum Einsatz kommen soll[1].

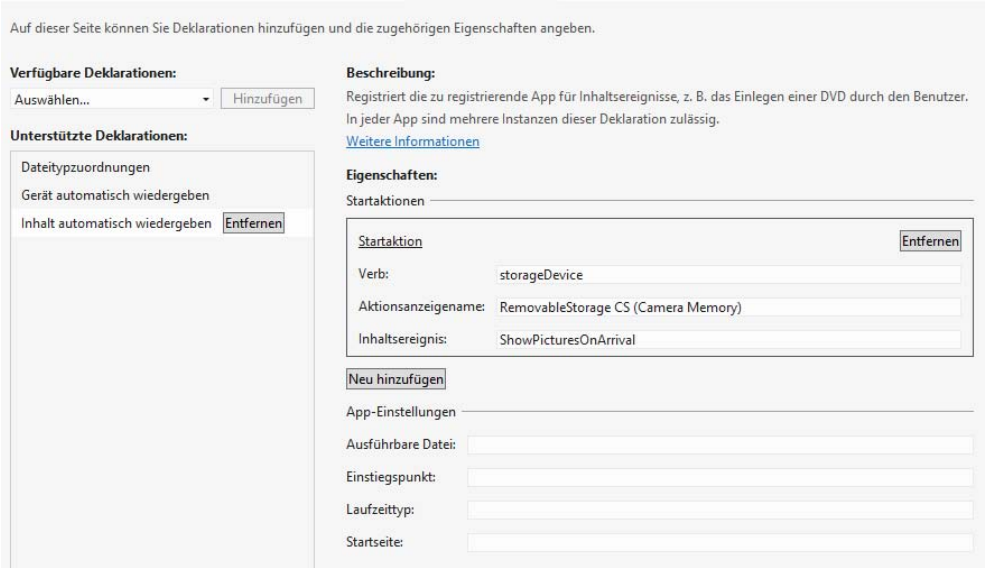
[1] NFC ist nicht ganz unumstritten, verzichten Sie also lieber darauf, wenn es nicht unbedingt sein muss.

Funktion	Beschreibung
Speicherort	Ermöglicht Ihrer App den Zugriff auf den aktuellen Standort des Tablets/Computers. Diese Information ist zum einen recht sensibel[1], zum anderen bietet sich hier aber auch die Möglichkeit, standortspezifische Programmfunktionen auszuführen.
Textnachrichten	Zugriff auf das Senden und Empfangen von Textnachrichten (SMS)
Unternehmensauthentifizierung	Verbindung mit Intranetressourcen per Domänenanmeldung
Wechselspeichergeräte	Zugriff auf USB-Sticks oder SD-Cards

HINWEIS: Unterlassen Sie es, die entsprechenden Funktionen zu deklarieren, so wird Ihre App teilweise mit Schutzverletzungen reagieren, wenn Sie trotzdem auf diese Funktionen zugreifen.

Deklarationen

Mit den Deklarationen definieren Sie die unterstützten Contracts und Extensions, gleichzeitig wird hier in vielen Fällen auch die Konfiguration vorgenommen.



In der obigen Abbildung sehen Sie beispielsweise, dass die App die automatische Wiedergabe von Medien unterstützt. Das *Verb* ist eine Kennung, die später beim Aufruf der Callback-Funktion an die App übertragen wird, um den Grund des Aufrufs zu spezifizieren. Das Inhaltsereignis definiert,

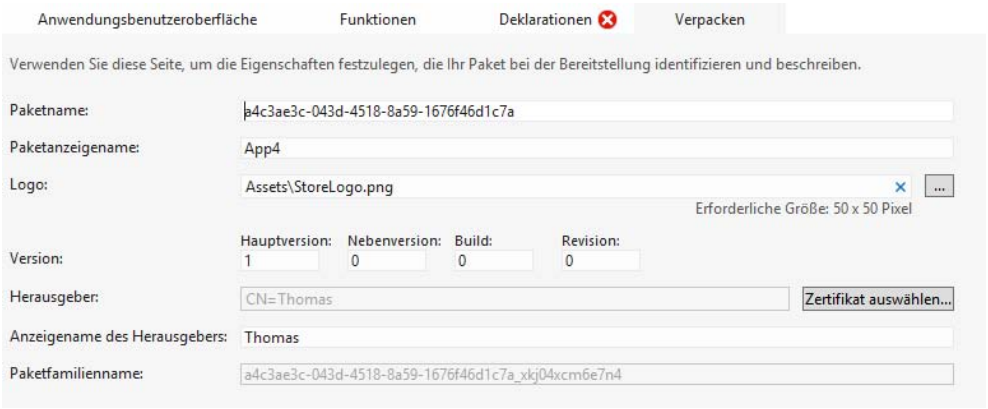
[1] Wer möchte schon auf Schritt und Tritt "beobachtet" werden?

wann das Ereignis ausgelöst wird, der Anzeigename erscheint beim Registrieren der App für die automatische Wiedergabe.

HINWEIS: An dieser Stelle werden wir nicht auf die verschiedenen Contracts eingehen, im Praxisteil dieses Kapitels finden Sie dazu zwei entsprechende Beispiele (Abschnitt 20.4.1 und 20.4.2).

Verpacken

Legen Sie hier die Optionen zum Verpacken der App fest. Dies betrifft vor allem den Anzeigenamen und das Zertifikat des Herausgebers:



20.1.6 Application1_TemporaryKey.pfx

Hierbei handelt es sich um ein selbst generiertes Zertifikat zum Signieren des App-Package. Jede App, die in der Windows 8-Oberfläche läuft, braucht ein gültiges Zertifikat.

HINWEIS: Obiges Zertifikat ist nur für die Entwicklung gültig, für eine finale Weitergabe ist ein Extra-Zertifikat erforderlich, das Sie sich bei einer Zertifizierungsstelle besorgen müssen.

20.1.7 MainPage.xaml & MainPage.xaml.cs

Dies ist die Startseite Ihrer Anwendung, d h. die erste Seite, die nach dem Splash Screen angezeigt wird. Welche Seite aus dem Projekt als Startseite fungiert legen Sie in der Datei *App.xaml.cs* fest. Siehe dazu auch

- 18.1.2 Die Page, der Frame und das Window
- 18.2 Projekttypen und Seitentemplates
- 18.3 Seitenauswahl und -navigation

20.1.8 Datentyp-Konverter/Hilfsklassen

Je nach Projekttyp werden Sie im Unterverzeichnis *Common* einen oder mehrere Konverter sowie Hilfsklassen vorfinden, die für die Datenbindung benötigt werden.

BooleanToVisibilityConverter.cs

Ein recht nützlicher Vertreter, der auch schon in WPF-Anwendungen gute Dienste geleistet hat, ist der *BooleanToVisibility*-Konverter. Da die Sichtbarkeit von Controls in XAML-Oberflächen nicht per *True* oder *False* gesteuert wird, sondern über die Eigenschaft *Visibility* (*Collapsed* oder *Visible*), ist beim Setzen der Eigenschaft mit einem booleschen Wert der Konverter unabdingbar.

```
public sealed class BooleanToVisibilityConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return (value is bool && (bool)value) ? Visibility.Visible : Visibility.Collapsed;
    }

    public object ConvertBack(object value, Type targetType, object parameter,
                              string language)
    {
        return value is Visibility && (Visibility)value == Visibility.Visible;
    }
}
```

BEISPIEL 20.3: Verwenden eines *Converter*s

C#

```
...
  <Page.Resources>
```

Hier wird eine Instanz des Konverters erzeugt ...

```
      <common:BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>
  </Page.Resources>
```

... die später im Rahmen der Datenbindung zum Einsatz kommt:

```
    <ItemsControl
                  x:Name="filtersItemsControl"
                  ItemsSource="{Binding Source={StaticResource filtersViewSource}}"
                  Visibility="{Binding ShowFilters,
                             Converter={StaticResource BooleanToVisibilityConverter}}"
                  Margin="120,-3,120,30">
...
```

BooleanNegationConverter.cs

Dieser Konverter negiert lediglich einen übergebenen booleschen Wert, aus *True* wird also *False*:

```
public sealed class BooleanNegationConverter : IValueConverter
{
```

```
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            return !(value is bool && (bool)value);
        }

        public object ConvertBack(object value, Type targetType, object parameter,
                                  string language)
        {
            return !(value is bool && (bool)value);
        }
    }
```

RichTextColumns.cs

Diese Hilfsklasse ermöglicht es, in der Detailansicht von Grid-Apps Texte in einem fixen Spaltenlayout darzustellen. Die Texte werden in fester Spaltenbreite entsprechend der verfügbaren Höhe formatiert, die Anzahl der erzeugten Spalten hängt von der Textlänge ab.

SuspensionManager.cs

Mit der Klasse *SuspensionManager* hatten Sie bereits im Einstiegsbeispiel (Abschnitt 17.3.5, ab Seite 882) erstmals zu tun. Aufgabe dieser Klasse ist das Abspeichern und spätere Wiederherstellen von Anwendungszuständen (Eingaben, aktive Seiten, aktives Control etc.), wenn die App von WinRT in den Suspend-Modus versetzt wird.

HINWEIS: Je nach Projekttyp wird die Klasse automatisch erzeugt, alternativ müssen Sie sich diese aus einem der vielen WinRT-Beispielprojekte kopieren.

Wie Sie die Klasse zum Speichern eigener Objekte einsetzen können, haben wir Ihnen bereits im Abschnitt 17.3.5 demonstriert.

Wesentlich einfacher ist es, wenn Sie nur einzelne Eigenschaften (z.B. den Inhalt einer *TextBox*) sichern und wiederherstellen möchten. Nutzen Sie die Seitenvorlage "Standardseite"(*BasicPage*), basiert diese auf der Klasse *LayoutAwarePage* und bringt bereits zwei vordefinierte Methodenrümpfe für das State-Management mit:

```
        protected override void LoadState(Object navigationParameter,
                                          Dictionary<String, Object> pageState)
        { }

        protected override void SaveState(Dictionary<String, Object> pageState)
        { }
```

Diese Methoden werden innerhalb der Klasse *LayoutAwarePage* durch die Navigation (*OnNavigatedTo* bzw. *OnNavigatedFrom*) aufgerufen. Intern nutzt dann *LayoutAwarePage* den Suspensionmanager, um die Dictionary-Daten der obigen Methoden seitenspezifisch zu sichern bzw. zu laden:

BEISPIEL 20.4: Sichern und Wiederherstellen des Inhaltes einer *TextBox*

C#

```
...
    protected override void LoadState(Object navigationParameter,
                                      Dictionary<String, Object> pageState)
        {
            if (pageState != null)
            {
                if (pageState.ContainsKey("txt1_txt")) TextBox1.Text =
                   (string)pageState["txt1_txt"];
            }
        }

        protected override void SaveState(Dictionary<String, Object> pageState)
        {
            pageState["txt1_txt"] = TextBox1.Text;
        }
```

Navigieren Sie von dieser Seite zu einer anderen Seite und kehren mit der Back-Taste zurück, wird der Inhalt der *TextBox* wieder hergestellt.

```
...
```

LayoutAwarePage.cs

Hierbei handelt es sich um die Basisklasse für die WinRT-Standardseiten. Diese Klasse implementiert Navigationsfunktionalität zwischen den einzelnen Seiten einer App (Back-Button, *GoHome*, *GoBack*, *GoForward*), unterstützt Tastenkombinationen und kümmert sich um das Sichern und Wiederherstellen von aktuellen Zuständen der Seite. Zusätzlich wird auch eine Basisfunktionalität für die visuellen Zustände der Seite implementiert. Mehr dazu unter

► 18.2.5 Standardseite (Basic Page)

HINWEIS: Diese Klasse wird automatisch in Ihr Projekt aufgenommen, wenn Sie darauf basierende Seitentemplates nutzen.

20.1.9 StandardStyles.xaml

Für die HTML-Programmierer sicherlich nichts Neues, für den ehemaligen Windows Forms-Programmierer hingegen Neuland ist die Datei *StandardStyles.xaml*.

Mit Hilfe der Styles lassen sich Eigenschaften für Controls bzw. Control-Gruppen auf eine elegante Art zentral verwalten. Zum einen müssen Sie bei einzelnen Controls nicht mehr umständlich einzelne Eigenschaften setzen, zum anderen können Sie jederzeit alle Controls, die auf einem Style basieren, mit einer neuen Optik ausstatten, indem Sie lediglich den Style ändern.

BEISPIEL 20.5: Verwendung *Styles*

XAML

Ein Style aus der Datei *StandardStyles.xaml*, der für *RichTextBlock*-Controls vorgesehen ist:

```
...
   <Style x:Key="BasicRichTextStyle" TargetType="RichTextBlock">
        <Setter Property="Foreground"
                Value="{StaticResource ApplicationForegroundThemeBrush}"/>
        <Setter Property="FontSize" Value="{StaticResource ControlContentThemeFontSize}"/>
        <Setter Property="FontFamily"
                Value="{StaticResource ContentControlThemeFontFamily}"/>
        <Setter Property="TextTrimming" Value="WordEllipsis"/>
        <Setter Property="TextWrapping" Value="Wrap"/>
        <Setter Property="Typography.StylisticSet20" Value="True"/>
        <Setter Property="Typography.DiscretionaryLigatures" Value="True"/>
        <Setter Property="Typography.CaseSensitiveForms" Value="True"/>
    </Style>
...
```

XAML

Die Verwendung:

```
<RichTextBlock Height="123" Width="298" Style="{StaticResource BaselineRichTextStyle}" />
```

Möchten Sie eigene Styles entwerfen, können Sie diese entweder

- in einer Seite definieren (dann ist dieser Style aber auch nur auf dieser Seite verfügbar),
- in einer eigenen Styledatei definieren (dann müssen Sie diese in die Datei *App.xaml* aufnehmen),
- oder direkt in der Datei A*pp.xaml* definieren.

Definieren Sie neue Styles, können diese auf bereits vorhandenen Styles basieren und entsprechend weiter konfiguriert werden:

BEISPIEL 20.6: Nutzen eines vorhanden *Styles* für einen neuen *Style*

XAML

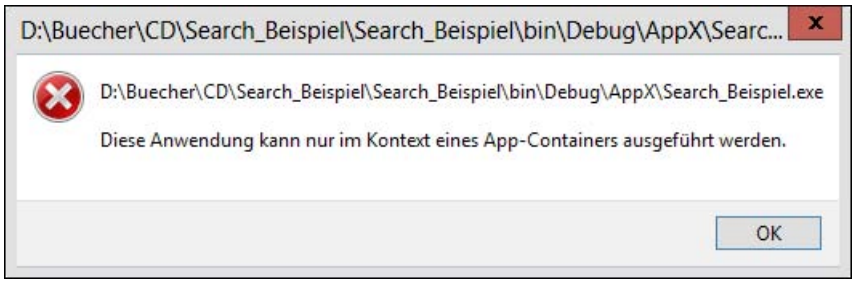
```
...
<Style x:Key="MyPageHeaderTextStyle" TargetType="TextBlock"
       BasedOn="{StaticResource HeaderTextStyle}">
    <Setter Property="Margin" Value="0,0,70,70"/>
</Style>
...
```

20.1.10 Assets/Symbole

Im Verzeichnis *Assets* finden Sie die im Zusammenhang mit dem Bereitstellen der App benötigten Symbole im PNG-Format. Prinzipiell sind die Namen frei wählbar, in diesem Fall müssen Sie aber die Manifestdatei *Package.appxmanifest* (siehe Seite 998) entsprechend anpassen. Achten Sie auf die Größe der einzelnen Dateien (Logo: 150x150 Pixel, Breites Logo 310x150 Pixel, Kleines Logo 30x30 Pixel, Splash Screen 620x300 Pixel).

20.1.11 Nach dem Kompilieren

Haben Sie Ihre App kompiliert, wollen Sie auch einen Blick in das Ausgabeverzeichnis *\\bin\Debug*. Neben diversen XML- und XAML-Dateien (Seiten, Manifest, App) finden sich hier auch die Ressourcen und eine EXE-Datei. Leider ist die Enttäuschung nach dem Start der EXE vorprogrammiert:



Der Grund für diesen Schock ist in der neuen Distribution von WinRT-Apps zu finden. Diese erfolgt im Normalfall über den Windows Store. Von dort heruntergeladene Apps bzw. App-Packages lassen sich dann in der Windows 8-Oberfläche installieren und werden später in einem App-Container ausgeführt.

HINWEIS: Bevor die App an den Microsoft-Store übertragen werden kann, muss diese in ein extra Package komprimiert werden. Nutzen Sie dazu den *Menüpunkt Projekt|Store|Anwendungspaket erstellen* (siehe Abschnitt 21.5 ab Seite 1118).

20.2 Der Lebenszyklus einer WinRT-App

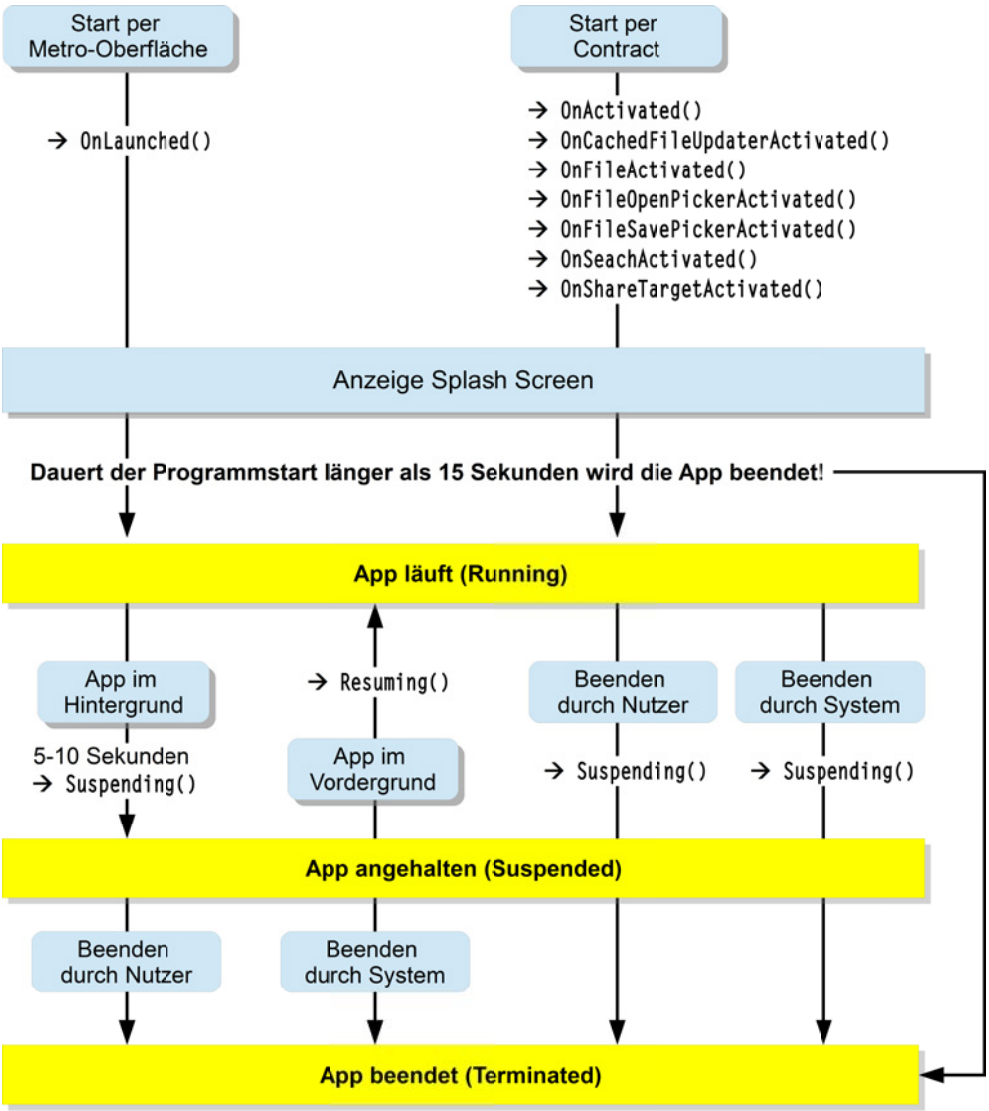
Wir hatten uns mit diesem Thema bereits an der einen oder anderen Stelle in den vorhergehenden Kapiteln kurz beschäftigt, hier soll es deshalb um einen zusammenfassenden Überblick gehen.

Wie schon erwähnt, werden für eine WinRT-App drei wesentliche "Zustände" unterschieden:

- Running
- Suspending
- Stopped

Ihre Aufgabe als Programmierer ist es, auf den Wechsel zwischen diesen Zustanden zu reagieren, um zum Beispiel den Status des Programms zu erhalten oder wieder herzustellen. Was sich so einfach anhört (mit Blick auf die bisher übliche Vorgehensweise in der Win32-Welt: Start → Programmablauf → Beenden) stellt Sie nun vor recht umfangreiche Probleme, die sich mit Hilfe einiger Methoden und Ereignisse jedoch lösen lassen.

Die folgende Abbildung versucht die komplexen Szenarien vereinfacht wiederzugeben:



Was wohl gleich ins Auge fällt sind drei grundsätzliche Unterschiede zu einem konventionellen Windows-Programm:

- die App kann aus vielen verschiedenen Gründen gestartet werden,
- die App wird durch das System in einen Ruhezustand versetzt,
- das Beenden der App ist aus Sicht des Programmierers unbestimmt.

Die folgenden Abschnitte sollen Ihnen Antworten auf sicherlich vorhandene Fragen geben.

20.2.1 Möglichkeiten der Aktivierung von Apps

Was sich zunächst etwas merkwürdig anhört, hat in einer WinRT-Umgebung dennoch seine Bedeutung: Ihre App kann nicht nur per Launch-Contract über die Windows 8-Oberfläche gestartet werden (dies ist für den Nutzer der zunächst offensichtlichste Weg), sondern auch durch eine Reihe von Contracts, wenn Ihre App diese registriert hat.

Leider hat diese Vielfalt auch einen Nachteil: Sie haben es nicht mit einem einheitlichen Eintrittspunkt in Ihre WinRT-App zu tun, sondern Sie müssen sich je nach Funktionsumfang mit mehreren Eintrittspunkten mit unterschiedlichen Parametern herumschlagen.

Standardmäßig haben Sie es beim Start über die Windows 8-Oberfläche mit der bereits in jedem Projekt (*App.xaml.cs*) angelegten *OnLaunched*-Methode zu tun, die leider etwas verwirrend ist:

```
protected override void OnLaunched(LaunchActivatedEventArgs args)
{
```

Versuch, den Frame des App-Windows zu ermitteln:

```
    Frame rootFrame = Window.Current.Content as Frame;
```

Ist dieser nicht bereits vorhanden, wird er erstellt:

```
    if (rootFrame == null)
    {
        rootFrame = new Frame();
```

Die App wurde vom System beendet:

```
        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
        }
```

Rahmen platzieren:

```
        Window.Current.Content = rootFrame;
    }
```

Ist keine Seite geladen:

```
    if (rootFrame.Content == null)
    {
```

Aufruf der Hauptseite mit Übergabe der Parameter:

```
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
        {
            throw new Exception("Failed to create initial page");
        }
    }
```

App-Window aktivieren:

```
    Window.Current.Activate();
}
```

Ganz uninteressant ist das obige Ereignis allerdings auch nicht, über die Argumente können Sie noch etwas mehr darüber in Erfahrung bringen.

Vergessen Sie an dieser Stelle jedoch *args.Kind*:

HINWEIS: Sie werden in dieser Methode immer nur den Wert *ActivationKind.Launch* erhalten, die Methode wird eben nur beim Start per Launch-Contract aufgerufen.

Ganz anders *args.PreviousExecutionState*: dieser Parameter gibt Ihnen Auskunft darüber, ob die App schon einmal lief und wenn ja, wie sie beendet wurde:

Wert	Vorheriger Zustand
NotRunning	■ die App ist neu installiert ■ die App wurde per Taskmanager beendet ■ der Computer wurde neu gestartet ■ der Anwender hat sich aus- und wieder eingeloggt
Running	In diesem Zusammenhang ohne Bedeutung, eine App kann nur einmal per Launch-Contract gestartet werden.
Suspended	In diesem Zusammenhang ohne Bedeutung, die Methode wird nur beim Start per Launch-Contract aufgerufen.
Terminated	Die App wurde durch das System beendet (z.B. wegen mangelnder Systemressourcen).
ClosedByUser	Die Anwendung wurde per Geste, Tastenkombination etc. vom Anwender beendet.

Auch wenn auf den ersten Blick zwei der Werte nie aufgerufen werden, behalten Sie diese dennoch im Auge. Nutzen Sie beispielsweise einen Search-Contract (siehe Beispiel in 20.4.1), wird die Methode *OnSearchActivated* mit den Parametern *args.Kind=Search* und *args.PreviousExecutionState=Running* aufgerufen. Womit wir bereits bei den anderen Startursachen für Ihre App angelangt sind. Je nach unterstütztem Contract wird statt der *OnLaunched*-Methode eine der folgenden Methoden aufgerufen:

- *OnActivated()*
- *OnCachedFileUpdaterActivated()*
- *OnFileActivated()*
- *OnFileOpenPickerActivated()*
- *OnFileSavePickerActivated()*
- *OnSeachActivated()*
- *OnShareTargetActivated()*

Auch hier können Sie über die Parameter zunächst unterscheiden, ob Ihre App bereits läuft oder ob Sie erst eine Seite bzw. den Frame initialisieren müssen. Zwei Anwendungen dafür finden Sie in den Praxisbeispielen

- 20.4.1 Unterstützung für den Search-Contract bieten
- 20.4.2 Die Auto-Play-Funktion unterstützen

Die Auswertung von *PreviousExecutionState* ist vor allem für das Laden vorhergehender Anwendungseinstellungen etc. interessant. So können Sie, je nach vorhergehendem Programmende, beispielsweise Daten laden oder neue Optionen setzen.

20.2.2 Der Splash Screen

Haben Sie sich die Abbildung zum Lebenszyklus einer WinRT-App auf Seite 1008 näher angesehen, so werden Sie sicher gemerkt haben, dass im Zusammenhang mit dem Start der App auch ein Splash Screen angezeigt wird. Dessen Inhalt besteht lediglich aus einer 620x300 Pixel großen PNG-Datei, die Sie in Ihr Projekt einbetten und im Anwendungsmanifest zuweisen.

HINWEIS: Die erste Seite Ihrer App sollte nach dem Start innerhalb von spätestens 15 Sekunden auf dem Bildschirm erscheinen, andernfalls wird Ihre App durch das System beendet. Der Splash Screen zählt dabei **nicht** als App-Seite, er wird vom System bereitgestellt.

Doch was tun, wenn Ihre App länger braucht? In diesem Fall "simulieren" Sie einfach den Splash Screen und zeigen quasi eine Kopie davon als erste Seite an (eventuell mit Fortschrittsanzeige). Wie es genau geht, zeigt das folgende Praxisbeispiel:

- 20.4.3 Einen zusätzlichen Splash Screen einsetzen

20.2.3 Suspending

Ist die App erst einmal gestartet und haben Sie eine Weile damit gearbeitet, so schalten Sie vermutlich zwischenzeitlich auch mal zu einer anderen App oder dem Windows-Dektop um. Auf diese Gelegenheit hat WinRT nur gewartet: schon nach wenigen Sekunden wird Ihre App in den Tiefschlaf versenkt (fast) nichts geht mehr[1]. Als Programmierer dürfte dies für Sie zunächst keine besondere Herausforderung sein, werden doch während dieses Zustands alle Programmfunktionen quasi eingefroren (keine CPU-Zeit, keine Ressourcen, Arbeitsspeicher bleibt jedoch belegt). Doch ganz so einfach ist es nicht – der Suspended-Status ist die letzte Stufe vor dem Programmende von der Sie etwas mitbekommen. Der Übergang von Suspended in Terminated ist für Sie nicht mehr transparent, stellen Sie sich also darauf ein, dass Ihr Programm aus dem Suspened-Mode nicht mehr "aufwacht".

Als Benachrichtigung, dass es jetzt an der Zeit ist den aktuellen Status der App zu sichern, dient Ihnen das *Suspending*-Ereignis, das bereits in jeder WinRT-App vordefiniert ist.

[1] Hintergrundprozesse werden natürlich noch weiter ablaufen.

```
        private void OnSuspending(object sender, SuspendingEventArgs e)
        {
            var deferral = e.SuspendingOperation.GetDeferral();
            // Code zum Sichern der App-Daten
            deferral.Complete();
        }
```

Über das *SuspendingDeferral*-Objekt können Sie WinRT mitteilen, dass Ihre App Daten sichert. Ist diese Aktion abgeschlossen, rufen Sie einfach die *Complete*-Methode auf.

HINWEIS: Beachten Sie, dass alle Aktionen innerhalb von fünf Sekunden abgeschlossen sein müssen, andernfalls wird der Vorgang zwangsweise abgebrochen.

Zum Sichern des App-Status nutzen Sie am besten gleich den *SuspensionManager* (siehe dazu Seite 1004). Doch was ist wenn Sie umfangreichere Daten sichern wollen? Hier hilft nur konsequentes Umdenken beim Programmablauf. Sichern Sie Daten nicht erst zu diesem Zeitpunkt, sondern dann wenn Sie anfallen (Eingaben, Seitenwechsel etc.).

Was eigentlich sollten Sie sichern? Je nach App kann es sinnvoll sein, die aktuelle Seite, die Eingabeposition und die Inhalte von Eingabemasken zu sichern (z.B. auch den Fortschritt von Wiedergaben).

HINWEIS: Diese Daten werden jedoch nur wieder hergestellt, wenn die App **neu** gestartet wird (*OnLaunched*, *args.PreviousExecutionState=Terminated*). Wird die App von WinRT lediglich reaktiviert (Resume) brauchen Sie sich **nicht** darum zu kümmern, die Daten sind ja noch alle aktuell. Wird *OnLaunched* mit einem anderen *PreviousExecutionState* aufgerufen, sollten Sie den vorhergehenden Programmstatus ebenfalls **nicht** wieder herstellen, es handelt sich ja quasi um einen Neustart der App.

Ganz nebenbei sollte Sie natürlich in obiger Methode alle Ressourcen freigeben, die dies erfordern (Dateien schließen, Verbindungen beenden, Geräte trennen). Bei einem Resume müssen Sie diese Aktionen dann auch wieder rückgängig machen, um den alten Status quo wieder herzustellen.

20.2.4 Resuming

Befindet sich Ihre App im angehaltenen Zustand (siehe auch Taskmanager) und wechselt der Nutzer zu dieser zurück, müssen Sie sich wohl oder übel auch mit dem "Aufwachen" der App beschäftigen. Dazu steht Ihnen das Ereignis *Resuming* zur Verfügung:

```
...
        public App()
        {
            this.InitializeComponent();
            this.Resuming += OnResuming;
        }
...
```

```
        static void OnResuming(object sender, object e)
        {
            Protokoll("OnResuming");
        }
```

In dieser Ereignisprozedur stellen Sie nur die Verbindungen zu Ressourcen (Dateien, Geräte etc.) wieder her, die Sie im *Suspending*-Ereignis getrennt haben. An dieser Stelle brauchen Sie keine Informationen zum Programmstatus wieder herzustellen, dafür ist im Falle eines Neustarts der App die *OnLaunched*-Methode zuständig. Eine kleine Ausnahme gibt es dennoch: War Ihre App länger nicht aktiv, sollten Sie überprüfen, ob alle Anzeigen (z.B. Datum/Zeit, Webdaten etc.) noch aktuell sind. Rechnen Sie auch damit, dass eventuell nicht alle bisher genutzten Ressourcen verfügbar sind.

20.2.5 Beenden von Apps

Tja, eigentlich sollte es diesen Abschnitt gar nicht geben. Apps werden prinzipiell nicht beendet, darum kümmert sich WinRT. Dennoch gibt es einige Gründe für ein Programmende:

- Mangel an Arbeitsspeicher[1],
- Usereingriff (Alt+F4 oder Wischgeste),
- das System wird heruntergefahren,
- die Anwendung bricht durch einen Fehler ab.

Prinzipiell gilt die Microsoft-Aussage: Es sollen keine wie auch immer gearteten Möglichkeiten zum aktiven Beenden der App im Nutzerinterface angeboten werden. Ob sich dies auch im weiteren Verlauf durchsetzt und praktikabel bleibt, sei dahingestellt. Eine kleine Ausnahme stellen diesbezüglich unsere Beispielprogramme dar, diese lassen sich über die *Beispiel beenden*-Schaltfläche sofort schließen. Der Vorteil für Sie als Entwickler: Sie müssen in Visual Studio nicht erst auf das von WinRT organisierten Programmende warten, sondern können gleich weiterarbeiten.

BEISPIEL 20.7: App per *Button* schließen (Achtung: nicht regelkonform!)

C#
```
...
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            Application.Current.Exit();
        }
...
```

Auf das Beenden der App können Sie nicht direkt per Ereignis oder Methode reagieren, Sie müssen schon vorher das *Suspendig*-Ereignis nutzen, um Ihre Anwendungsdaten zu sichern. Das Beenden der App per Alt+F4 oder Wischgeste führt zwischenzeitlich auch zu einem *Suspending*-Aufruf, (siehe Abbildung auf Seite 1008), Sie können also Ihre Daten auch in diesem Fall problemlos sichern.

[1] Ja, auch das soll es auch noch geben!

20.2.6 Die Ausnahmen von der Regel

In den bisherigen Ausführungen haben Sie sicher einige Aufgaben vermisst:

- Musikwiedergabe (im Hintergrund)
- Datenübertragung von größeren Dateien
- Reaktion auf Systemereignisse

Alle diese Aufgaben erfordern einen Eingriff in die regulären WinRT-Verwaltungsaufgaben, den keine dieser Aufgaben soll durch den Wechsel der App (dies zieht ein *Suspending* nach sich) beendet oder unterbrochen werden.

HINWEIS: Keine Hintergrundaufgabe hat die Möglichkeit, das Nutzerinterface Ihrer App zu aktualisieren. Sie müssen sich auf Popupmeldungen oder das Aktualisieren der App-Kachel beschränken.

20.2.7 Debuggen

Oh je, ein in diesem Zusammenhang recht diffiziles Thema stellt das Debuggen der Zustände einer App dar. Grundsätzlich gilt:

HINWEIS: Testen Sie Ihre App möglichst **ohne** den Visual Studio-Debugger, dieser beeinflusst die Task-Verwaltung von WinRT teilweise recht umfangreich (z.B. Timeouts, Zustandsanzeige im Taskmanager) und Sie erzielen keine realistischen Ergebnisse.

Nutzen Sie deshalb entweder

- eine Protokolldatei, in der Sie Informationen mit Zeitmarken abspeichern oder
- verwenden Sie das Programm *DebugView* zur Anzeige der Debug-Messages.

Verwendung einer Protokolldatei

Wer seine App bei externen Testern prüfen lässt ist gut beraten, Debug-Ausgaben in einer Protokoll-Datei zu sichern und diese bei Gelegenheit zu prüfen. Das folgenden Beispiel zeigt eine einfache Lösung:

BEISPIEL 20.8: Möglichkeit zum Speichern einer Protokolldatei

C#

Fügen Sie zunächst in der Manifestdatei die Funktion "Zugriff auf Dokumentenbibliothek" ein. Nachfolgend benötigen Sie noch eine Deklaration für eine "Dateitypzuordnung". Vergeben Sie als Namen "text" und als Dateityp ".txt".

Erweitern Sie die Definition der *App*-Klasse um folgende Einträge:

```
...
using Windows.Storage;
```

BEISPIEL 20.8: Möglichkeit zum Speichern einer Protokolldatei

```
...
    sealed partial class App : Application
    {
```

Die Datei, in der wir die Daten speichern:

```
        private static StorageFile _file;
```

Ein Zwischenspeicher für die Protokolldaten:

```
        private static List<string> _protokoll = new List<string>();
...
```

Zum Füllen der Protokollliste nutzen wir folgende Funktion:

```
        private static void Protokoll(string msg)
        {
            _protokoll.Add(DateTime.Now.ToString() + ": " + msg);
        }
```

Das eigentliche Sichern der Protokolldaten:

```
        async private static void SaveProtokoll()
        {
            _file = await KnownFolders.DocumentsLibrary.CreateFileAsync("MeinLogfile.txt",
                              CreationCollisionOption.OpenIfExists);
            await FileIO.AppendLinesAsync(_file, _protokoll);
            _protokoll.Clear();
        }
```

Mit dem folgenden Aufruf wird das Protokoll gesichert:

```
       async private void OnSuspending(object sender, SuspendingEventArgs e)
        {
            var deferral = e.SuspendingOperation.GetDeferral();
            Protokoll("OnSuspending");
            SaveProtokoll();
            deferral.Complete();
        }
...
```

Ergebnis

```
09.08.2012 09:57:42: OnLaunched Kind=Launch  PreviousExecutionState=ClosedByUser
09.08.2012 09:57:42: Window.Current.Activated
09.08.2012 09:57:52: Window.Current.Activated
09.08.2012 09:57:52: OnSearchActivated Kind=Search  PreviousExecutionState=Running
09.08.2012 09:57:52: Window.Current.Activated
09.08.2012 09:57:56: Window.Current.Activated
09.08.2012 09:57:57: Window.Current.VisibilityChanged
09.08.2012 09:58:07: OnSuspending
09.08.2012 10:55:05: OnLaunched Kind=Launch  PreviousExecutionState=NotRunning
09.08.2012 10:55:05: Window.Current.Activated
09.08.2012 10:55:06: Window.Current.Activated
```

BEISPIEL 20.8: Möglichkeit zum Speichern einer Protokolldatei

```
09.08.2012 10:55:07: Window.Current.VisibilityChanged
09.08.2012 10:55:17: OnSuspending
...
```

HINWEIS: Alternativ könnten Sie sich natürlich auch damit beschäftigen, die Daten gleich per E-Mail oder Webdienst zu versenden.

Debuggen mit DebugView

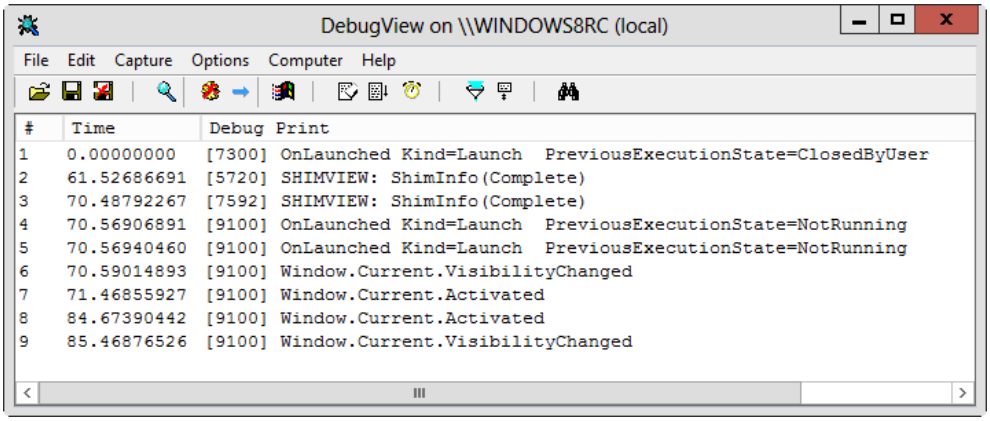
Sind Sie es leid, Ihre App unter den Einschränkungen des Visual Studio-Debuggers zu testen, nutzen Sie doch einfach *DebugView*, das in seiner aktuellen Version auch in der Lage ist, Debug-Messages (*Debug.WriteLine*) abzufangen und anzuzeigen.

Herunterladen können Sie das Tool unter der folgenden Adresse:

LINK: `http://technet microsoft.com/en-us/sysinternals/bb896647`

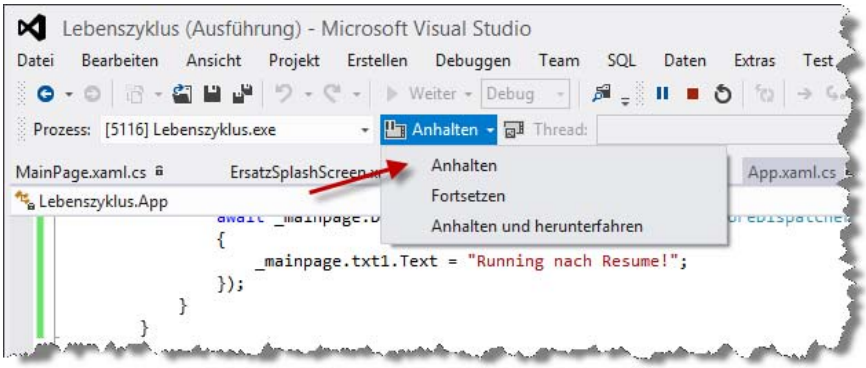
Entpacken Sie die enthaltenen Daten und starten Sie das Programm. Verteilen Sie in Ihrer App an den gewünschten Stellen die Debug-Aufrufe und starten Sie die App zunächst einmal aus Visual Studio heraus, um diese zu installieren. Beenden Sie dann die App und starten Sie aus der Windows 8-Oberfläche heraus.

Eine Beispielausgabe:



Steuern der Programmzustände über Visual Studio

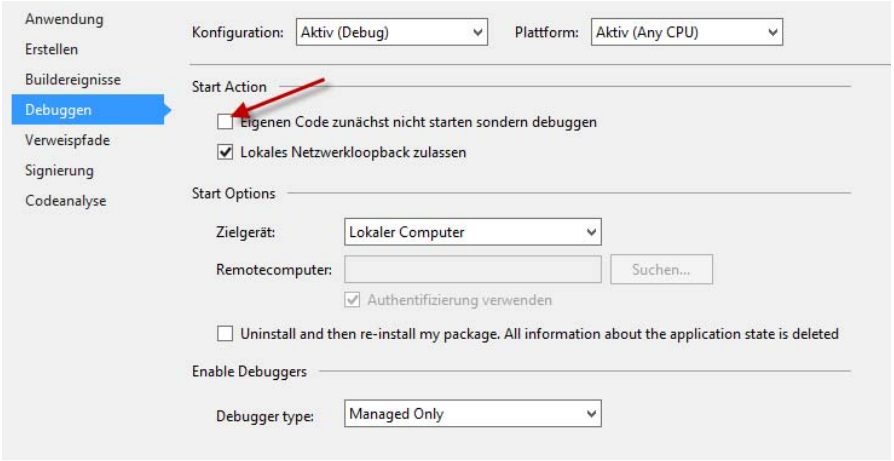
"Zeit ist Geld" und so haben Sie sicher nicht immer Lust zu warten, bis WinRT der Meinung ist, Ihre App in den Suspend-Modus zu schicken (ganz abgesehen davon, dass dies im Debug-Modus im Normalfall nicht klappt). Aus diesem Grund bietet auch Visual Studio Möglichkeiten, die App zu beeinflussen und z.B. in den Suspended-Modus schicken:



HINWEIS: Doch Achtung: Sie erhalten im Taskmanager kein Feedback, dass die App wirklich im Suspended-Modus ist. Das Ereignis *Suspending* wird hingegen ausgelöst.

Debuggen von Contract-aktivierten Apps

Soll Ihre App einen Contract unterstützen, ist es mitunter lästig, wenn im Debug-Mode die App sofort per Launch-Contract startet (sie soll ja eigentlich erst erscheinen, wenn sie vom System aufgerufen wird). So zum Beispiel auch bei Verwendung eines Search-Contracts: Im Normalfall sollte der erste Aufruf im Programm die *OnSearchActivated*-Methode sein, im Debug-Modus startet aber immer erst *OnLaunched*. Auch hier wurde von den Microsoft-Entwicklern eine Lösung implementiert. Setzen Sie in den Debug-Optionen (*Projekt|Eigenschaften|Debuggen*) einfach ein Häkchen bei "Eigenen Code zunächst nicht starten sondern debuggen":



Nachfolgend wird zwar Ihre App installiert und es startet der Debugger, aber die eigentliche App wird noch nicht aufgerufen. Erst mit Aufruf eines Contracts (z.B. Search) startet auch die App.

Damit schließen wir die Ausführungen zum Lebenszyklus von WinRT-Apps ab und wenden uns einem neuen Thema zu.

20.3 Daten speichern und laden

Erwarten Sie jetzt bitte keine weitschweifigen Ausführungen zu Datenbanken etc., wir wollen uns an dieser Stelle ganz trivial mit den Möglichkeiten beschäftigen, wie Sie Programminformationen persistent auf dem Anwender-PC oder auch in der Cloud ablegen können.

Einige wenige Beispiele hatten Sie ja schon in den vorhergehenden Kapiteln kennen gelernt, nun wollen wir das Ganze noch einmal zusammenhängend betrachten.

20.3.1 Grundsätzliche Überlegungen

Bevor Sie jetzt an wohlvertraute und bekannten Konzepte denken, sollten Sie zunächst überlegen, ob sich unter Windows 8 nicht andere Möglichkeiten für Ihre App ergeben.

Prinzipiell sollten Sie immer im Auge behalten, dass Ihre App vom Nutzer auch auf mehreren (auch mobilen) Geräten genutzt werden kann. Die bedeutet gegenüber der bisherigen Vorgehensweise – ein Programm → einmal Daten – ein komplettes Umdenken. Sie müssen zwischen Informationen unterscheiden die

- nur lokal,
- oder lokal und auf möglichen anderen Geräten

genutzt werden sollen. Es mach ja wohl kaum einen Sinn, dass der Nutzer z.B. auf allen Geräten seine Lesezeichen, Kontaktdaten oder Grundeinstellungen neu eingibt, wenn er mit derselben App arbeiten will. Im Gegensatz dazu sollen natürlich nicht immer die Eingaben der einen App auch auf allen Geräten verfügbar sein.

20.3.2 Worauf und wie kann ich zugreifen?

Diese Fragen sind für den bisherigen Desktop-Programmierer auf den ersten Blick nicht ganz einsichtig. Wenn Sie sich jedoch an das Sandbox-Prinzip für WinRT-Apps zurück erinnern, wurde in diesem Zusammenhang auch von einer drastischen Einschränkung beim Dateizugriff gesprochen. Prinzipiell sind folgende Datenquellen/-verzeichnisse ganz oder teilweise für Ihre App verfügbar:

- AppData-Verzeichnis
- lokale Daten
- Netzwerk- oder Geräte-Daten
- Web-Daten

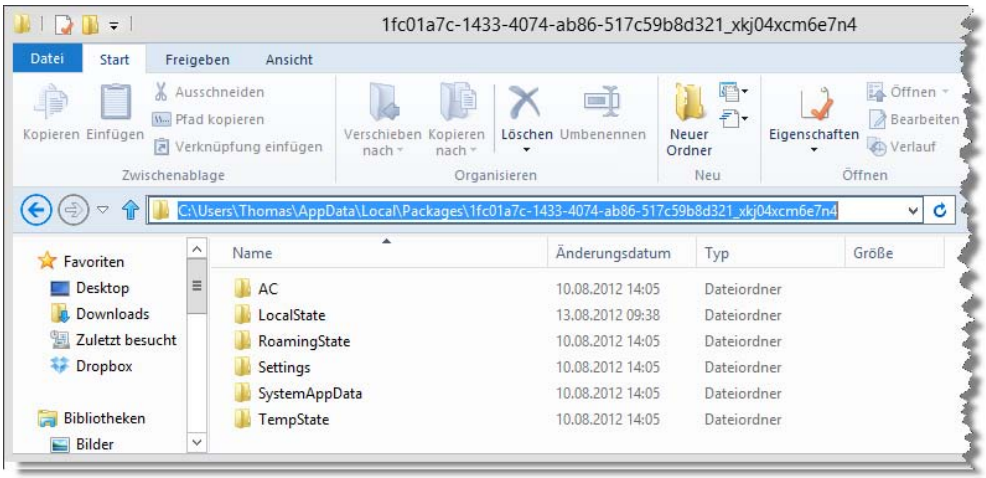
Sehen wir uns die einzelnen Verzeichnisse einmal im Detail an.

20.3.3 Das AppData-Verzeichnis

Hauptanlaufpunkt Ihrer App ist das programmspezifische *AppData*-Verzeichnis, zu finden unter

```
C:\Users\<Anwendername>\AppData\Local\Packages\<Package-ID>
```

HINWEIS: Die Package-ID können Sie der Manifestdatei entnehmen (Seite *Verpacken*) .



LocalFolder

Auf den Unterordner *LocalState* haben Sie unbeschränkten Zugriff, Sie können hier Verzeichnisse und Dateien schreiben, lesen und löschen, so wie Sie es auch von Desktop-Anwendungen gewohnt sind.

HINWEIS: Sie sind an dieser Stelle nicht auf das Wohlwollen des Anwenders angewiesen, Sie können ohne Nachfrage (oder andere Dialoge) speichern!

BEISPIEL 20.9: Erzeugen einer Datei im Unterordner *LocalState*

C#

```
...
using Windows.Storage;
...
        async private void Button_Click_7(object sender, RoutedEventArgs e)
        {
            var localFolder = ApplicationData.Current.LocalFolder;
            var file = await localFolder.CreateFileAsync("Testdatei.txt");
        }
...
```

Doch Vorsicht:

HINWEIS: Daten, die Sie hier ablegen, sind zwar vor dem Zugriff durch andere WinRT-Apps geschützt, der Nutzer hat jedoch unbeschränkten Zugriff auf dieses Verzeichnis. Speichern Sie hier also Daten für Spielstände, Restriktionen etc. ab, können diese jederzeit durch den Anwender verändert oder gelöscht werden.

Bleibt die Frage, wie Sie per XAML auf Daten in diesem Verzeichnis zugreifen können:

BEISPIEL 20.10: Einbinden von Bildern aus dem ||*AppData*|*LocalState*-Verzeichnis

XAML
```
...
  <Image Source="ms-appdata:///local/hintergrund.png" Stretch="Fill"/>
...
```

RoamingFolder

Wie Sie der Abbildung auf der vorhergehenden Seite entnehmen können, findet sich unter den Anwendungsdaten auch ein Verzeichnis *RoamingState*, in dem Daten abgelegt werden, die mit Hilfe des Ihres LiveID-Kontos zwischen mehren PCs abgeglichen werden.

HINWEIS: Dieser Ordner ist im Gegensatz zu den reinen lokalen Daten etwas eingeschränkt. So ist die Verzeichnisgröße auf den Wert von *ApplicationData.Current.Roaming-StorageQuota* begrenzt. Speichern Sie mehr Daten als erlaubt, stellt das System den Datenabgleich über die Cloud ein.

BEISPIEL 20.11: Erzeugen einer Roaming-Datei im Unterordner *RoamingState*

C#
```
...
        async private void Button_Click_8(object sender, RoutedEventArgs e)
        {
            var roamingFolder = ApplicationData.Current.RoamingFolder;
            var file = await roamingFolder.CreateFileAsync("Testdatei.txt",
                            CreationCollisionOption.ReplaceExisting);
            ApplicationData.Current.SignalDataChanged();
        }
...
```

Der Zugriff per XAML:

```
<Image Source="ms-appdata:///roaming/hintergrund.png"/>
```

TemporaryFolder

Auch an das Ablegen temporärer Daten wurde gedacht: diese finden Sie im *TempState*-Unterordner des App-Datenverzeichnisses.

HINWEIS: Dateien in diesem Verzeichnis können jederzeit durch den Anwender (Datenträgerbereinigung) oder das System gelöscht werden.

BEISPIEL 20.12: Erzeugen einer temporären Datei im Unterordner *TempState*

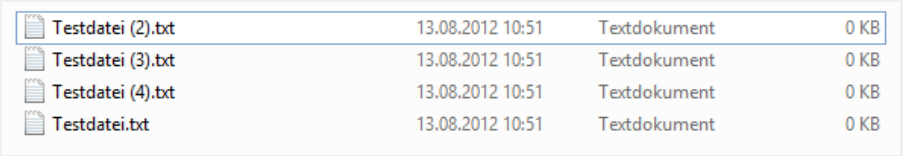
C#
```
...
        async private void Button_Click_8(object sender, RoutedEventArgs e)
        {
            var tempFolder = ApplicationData.Current.TemporaryFolder;
```

BEISPIEL 20.12: Erzeugen einer temporären Datei im Unterordner *TempState*

```
            var file = await tempFolder.CreateFileAsync("Testdatei.txt",
                                CreationCollisionOption.GenerateUniqueName);
        }
...
```

Ergebnis

Laufzeitansicht nachdem obige Methode mehrfach aufgerufen wurde:



Auch hier können Sie per XAML auf die Datei verweisen, was jedoch nicht unbedingt empfehlenswert ist:

```
<Image Source="ms-appdata:///temp/hintergrund.png"/>
```

20.3.4 Das Anwendungs-Installationsverzeichnis

Neben dem App-Datenverzeichnis besitzt jede App auch ein spezifisches Installationsverzeichnis, das Sie unter folgendem Pfad finden[1]:

```
C:\Program Files\WindowsApps\<Package-ID>
```

Hier haben Sie lesenden Zugriff auf alle Dateien Ihres Installations-Packages.

BEISPIEL 20.13: Inhalte des Installationsverzeichnisses auflisten

C#

```
...
        async private void Button_Click_9(object sender, RoutedEventArgs e)
        {
            StorageFolder installlocFolder =
                    Windows.ApplicationModel.Package.Current.InstalledLocation;
            ListBox1.Items.Add(installlocFolder.Path);
            foreach (var file in await installlocFolder.GetFilesAsync())
                ListBox1.Items.Add("    -->" + file.Path);
            foreach (var folder in await installlocFolder.GetFoldersAsync())
            {
                ListBox1.Items.Add(folder.Path);
                foreach (var file in await folder.GetFilesAsync())
                    ListBox1.Items.Add("    -->" + file.Path);
            }
        }
...
```

[1] Dies gilt nur bei einer ordnungsgemäßen Installation per Store oder PowerShell-Skript.

Der Zugriff per XAML:

```
<Image Source="ms-appx:///myFile.png"/>
```

20.3.5 Das Downloads-Verzeichnis

Hier haben wir es mit einem Novum zu tun: dieses Verzeichnis ist nicht schreib-, sondern lesegeschützt. Sie haben also keine Möglichkeit, nach dem Schreiben einer Datei bzw. eines Verzeichnisses auf dieses zuzugreifen.

So können Sie zwar ein Verzeichnis erzeugen:

```
...
var folder = await DownloadsFolder.CreateFolderAsync("Testen");
...
```

Versuchen Sie aber dieses Verzeichnis später erneut zu öffnen

```
var folder = await DownloadsFolder.CreateFolderAsync("Testen",
                                  CreationCollisionOption.OpenIfExists);
```

werden Sie darüber belehrt, dass diese Option nicht zulässig ist (nur *FailIfExists* und *GenerateUniqueName*). Sie können also bei einem späteren Aufruf nicht mehr auf diesen Order zugreifen.

Gleiches gilt für eine erzeugte Datei: Solange Sie diese nicht freigeben, können Sie alles mit dieser Datei anstellen. Nach einem *Dispose* haben Sie aber keinen Zugriff mehr.

BEISPIEL 20.14: Datei im Downloads-Verzeichnis anlegen

C#

```
...
using Windows.Storage;
...
        async private void Button_Click_10(object sender, RoutedEventArgs e)
        {
```

Unterordner erzeugen:

```
            var folder = await DownloadsFolder.CreateFolderAsync("Testen",
                                    CreationCollisionOption.GenerateUniqueName);
```

Datei in den Unterordner schreiben:

```
            var file = await folder.CreateFileAsync("Testdatei.txt",
                                    CreationCollisionOption.GenerateUniqueName);
```

Datei direkt in *Downloads* schreiben:

```
            var file2 = await DownloadsFolder.CreateFileAsync("Testdatei.txt",
                                    CreationCollisionOption.GenerateUniqueName);
        }
...
```

20.3.6 Sonstige Verzeichnisse

Neben den bereits genannten Verzeichnissen gibt es noch weitere Verzeichnisse, auf die Sie Zugriff erlangen können. Allerdings müssen Sie dazu vorher die entsprechenden Rechte per Manifest und Contract (siehe dazu 20.1.1 ab Seite 994 und Abschnitt 20.1.5 Seite 998) einholen.

Haben Sie die Funktionen eingebunden, stellt sich die Frage, wie Sie den Pfad zu diesen Verzeichnissen ermitteln. Hier hilft Ihnen die *Windows.Storage.KnownFolders*-Klasse mit den entsprechenden Methoden weiter:

- *DocumentsLibrary*
- *HomeGroup*
- *MediaServerDevices*
- *MusicLibrary*
- *PicturesLibrary*
- *RemovableDevices*
- *VideosLibrary*

BEISPIEL 20.15: Alle Bilder in *Bibliotheken|Bilder* auflisten

C#
```
...
        async private void Button_Click_11(object sender, RoutedEventArgs e)
        {
            StorageFolder pictures = KnownFolders.PicturesLibrary;
            foreach (var file in await pictures.GetFilesAsync())
                ListBox1.Items.Add("    -->" + file.Path);
        }
...
```

HINWEIS: Vergessen Sie nicht die entsprechenden Funktionen freizuschalten, ansonsten wird der Zugriff blockiert!

20.3.7 Anwendungsdaten lokal sichern und laden

Nachdem wir uns nun mit den verschiedenen Speicherorten vertraut gemacht haben, wollen wir uns ansehen, wie wir Anwendungsdaten auf dem Anwender-PC/-Tablet sichern und wieder abrufen können. Eine erste Möglichkeit für das Sichern von lokalen Dateien hat Ihnen der Abschnitt 20.3.3 bereits aufgezeigt. Doch was ist, wenn es sich nur um einzelne Variablen oder Eigenschaften handelt? Hier helfen Ihnen die Anwendungseinstellungen (Klasse *ApplicationDataContainer*) weiter.

HINWEIS: In den Anwendungseinstellungen sollten Sie in keinem Fall Daten sichern, die für den Anwender dauerhaft von Interesse sind. Wird die App deinstalliert, gehen diese Informationen unwiederbringlich verloren. Derartige Daten sollten nach wie vor im normalen Dateisystem (z.B. *\\Dokumente* oder SkyDrive) gespeichert werden.

Schauen wir uns also an, wie verschiedene Arten von Daten mit Hilfe der Anwendungseinstellungen gesichert werden können.

BEISPIEL 20.16: Einfache Werte sichern und laden

C#

```
...
using Windows.Storage;
...
    ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;
```

Einen Wert schreiben:

```
    localSettings.Values["EinfacherWert"] = "Ein gespeicherter Wert!";
```

Den Wert später wieder lesen:

```
    ListBox1.Items.Add((string)localSettings.Values["EinfacherWert"]);
...
```

Für die meisten Anwendungsfälle dürfte obige Variante ausreichen. Was aber ist, wenn sich Daten kontinuierlich ändern und Sie sicherstellen wollen, dass zusammengehörige Daten auch konsistent gespeichert werden?

Hier hilft Ihnen ein zusammengesetzter Wert, der in "einem Rutsch" gespeichert und geladen wird:

BEISPIEL 20.17: Zusammengesetzte Werte speichern

C#

```
...
     ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;
     ApplicationDataCompositeValue composite = new ApplicationDataCompositeValue();
     composite["Username"] = "Müller";
     composite["Punktestand"] = (int)1000;
```

Zusammengesetzten Wert speichern:

```
     localSettings.Values["BeispielCompositeValue"] = composite;
```

Zusammengesetzten Wert laden:

```
      composite=(ApplicationDataCompositeValue)
                 localSettings.Values["BeispielCompositeValue"];
      ListBox1.Items.Add((string) composite["Username"]);
      ListBox1.Items.Add((int)composite["Punktestand"]);
...
```

Haben Sie große Datenmengen zu verwalten, können Sie die Daten auch in Containern zusammenfassen. Diese lassen sich dann später auch in einem Rutsch "entsorgen".

BEISPIEL 20.18: Werte in Containern speichern

C#

```
...
    ApplicationDataContainer localSettings = ApplicationData.Current.LocalSettings;
```

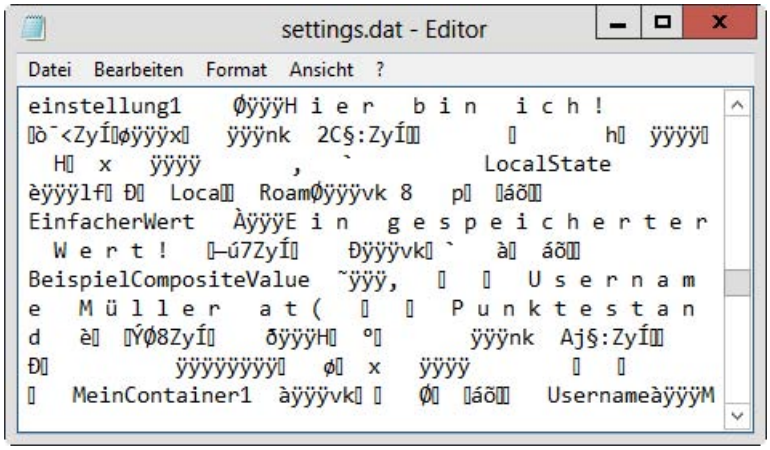
Container erstellen:

```
    ApplicationDataContainer container = localSettings.CreateContainer("MeinContainer1",
                                         ApplicationDataCreateDisposition.Always);
    if (localSettings.Containers.ContainsKey("MeinContainer1"))
    {
```

Daten in den Container schreiben:

```
        localSettings.Containers["MeinContainer1"].Values["Username"] = "Müller";
    }
...
```

All diese Daten werden in der Datei *settings.dat* im Unterverzeichnis *C:\Users\<Username>\AppData\Local\Packages\<Package-ID>\Settings* gesichert. Es handelt sich zwar nicht um eine XML-Datei, der Inhalt ist aber (mit etwas gutem Willen) auch so lesbar:



20.3.8 Daten in der Cloud ablegen/laden (Roaming)

Wie schon erwähnt, bieten Windows 8-Apps auch die Möglichkeit, Daten per Roaming über die "LiveId" des Anwenders zwischen mehreren Geräten zu synchronisieren, vorausgesetzt, der Anwender ist per LiveId angemeldet und die entsprechende Option ist in den Systemeinstellungen aktiviert:

Dies betrifft sowohl komplette Dateien (siehe "RoamingFolder" ab Seite 1020) als auch einzelne Einstellungen. Diese werden ebenfalls in der Datei *settings.dat* im Unterverzeichnis

```
C:\Users\<Username>\AppData\Local\Packages\<Package-ID>\Settings
```

zwischengespeichert.

Die Unterschiede zum lokalen Speichern der Daten sind aus Entwicklersicht zunächst minimal: Statt

```
ApplicationData.Current.LocalSettings
```

nutzen Sie jetzt

```
ApplicationData.Current.RoamingSettings
```

Alle Varianten, wie im vorhergehenden Abschnitt gezeigt, stehen Ihnen auch hier zur Verfügung. Mit einem Unterschied: die maximale Datenmenge für den Abgleich zwischen den einzelnen Clients beträgt derzeit 100 KByte. Überschreiten Sie diesen Wert, werden die Daten nicht mehr abgeglichen.

BEISPIEL 20.19: Speichern und Laden eines Wertes per Roaming

C#

```
...
      ApplicationDataContainer roaming = ApplicationData.Current.RoamingSettings;
      roaming.Values["einstellung1"] = "Hier bin ich!";
```

BEISPIEL 20.19: Speichern und Laden eines Wertes per Roaming

```
...
    ApplicationDataContainer roaming = ApplicationData.Current.RoamingSettings;
    if (roaming.Values["einstellung1"]!=null)
       ListBox1.Items.Add((string) roaming.Values["einstellung1"]);
...
```

Das gültige Maximum für die synchronisierbare Datenmenge erfahren Sie über *ApplicationData.Current.RoamingStorageQuota*, den aktuellen "Füllstand" können Sie derzeit nicht ermitteln[1].

Möchten Sie auf Änderungen der Daten reagieren, sollten Sie das *DataChanged*-Ereignis unterstützen. Damit ist Ihre App immer auf dem aktuellen Stand, auch wenn ein anderer Client die Daten ändert.

BEISPIEL 20.20: Reaktion auf das *DataChanged*-Ereignis

C#

```
...
        public BasicPage1()
        {
            this.InitializeComponent();
            ApplicationData.Current.DataChanged +=Current_DataChanged;
        }

        async void Current_DataChanged(ApplicationData sender, object args)
        {
```

Achtung: Der Aufruf erfolgt aus einem anderen Kontext heraus, Sie müssen zum aktuellen Thread wechseln:

```
          await this.Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,() =>
          {
                ListBox1.Items.Add("Daten wurden geändert!!!!");
                ApplicationDataContainer roaming =
                          Windows.Storage.ApplicationData.Current.RoamingSettings;
                if (roaming.Values["einstellung1"] != null)
                    ListBox1.Items.Add((string)roaming.Values["einstellung1"]);
          });
        }
...
```

HINWEIS: Versprechen Sie sich jedoch nicht allzu viel von dieser Funktion, die Daten werden nur recht sporadisch aktualisiert. Es geht hier nicht um Sekunden oder Minuten, sondern es kann auch schon mal eine halbe Stunde dauern.

[1] Es gab in der Beta mal die Eigenschaft *ApplicationData.Current.RoamingStorageUsage* ...

20.3.9 Aufräumen

Ein unter Programmierern recht unbeliebtes Thema, wie man nach der Deinstallation vieler Programme feststellen muss. Aus diesem Grund hat sich Microsoft der Sache konsequent angenommen:

HINWEIS: Wird eine App deinstalliert, werden auch alle Programmeinstellungen gelöscht, d.h., das komplette Verzeichnis *C:\Users\<Anwendername>\AppData\Local\Packages\<Package-ID>*.

So weit so gut, doch was ist, wenn Sie nur einzelne Einstellungen löschen möchten um z.B. neugierigen Blicken vorzubeugen? Hier helfen Ihnen die folgenden drei Varianten weiter:

- *localSettings.**Values.Remove**("EinfacherWert")*
- *localSettings.**Values.Clear**()*
- *localSettings.**DeleteContainer**("MeinContainer1")*

Letzteres löscht eine ganze Gruppe von Einzeleinträgen.

Die gleiche Vorgehensweise gilt sinngemäß auch für die Roaming-Daten, was besonders wichtig ist. Löschen Sie beispielsweise eine App auf einem PC und installieren Sie diese erneut, werden andernfalls die Roamingdaten aus der Cloud wieder hergestellt, was nicht immer beabsichtigt ist.

HINWEIS: Wie Sie Anwendungsdateien und Verzeichnisse löschen können, zeigt Ihnen der Abschnitt 21.1 ab Seite 1049.

20.3.10 Sensible Informationen speichern

An der einen oder anderen Stelle haben wir Sie ja schon darauf aufmerksam gemacht: die Anwendungseinstellungen sind zwar für andere WinRT-Apps nicht sichtbar, für den Anwender oder andere Desktop-Programme stellt es jedoch kein Hindernis dar, die App-Daten zu lesen oder gar zu verändern. Damit dürfte diese Variante für das Sichern von Credentials etc. natürlich ausfallen.

Genau für diesen Einsatzzweck haben die Microsoft-Programmierer die *PasswordVault*-Klasse entwickelt. Diese bietet nicht nur die Möglichkeit, eine Name/Passwort-Kombination verschlüsselt zu speichern. Arbeiten Sie mit einer LiveId-Anmeldung und ist die entsprechende Funktion in den Systemeinstellungen freigeschaltet, können Anmeldedaten über mehrere Geräte geteilt werden.

BEISPIEL 20.21: Credentials speichern, laden und löschen

C#

```
...
using Windows.Security.Credentials;
...
```

Die zu speichernden Daten sollten aus Eingabefeldern stammen. Neben dem spezifischen Ressourcenamen (diesen legen Sie fest) wird noch die altbekannte Name/Passwort-Kombination übergeben:

BEISPIEL 20.21: Credentials speichern, laden und löschen

```
    new PasswordVault().Add(new PasswordCredential("MyServer1", "MayerMüller", "geheim"));
...
```

Laden der Daten:

```
...
            PasswordVault vault = new PasswordVault();
            try
            {
```

Wir rufen zunächst die Ressource über ihren Namen ab:

```
                var creds = vault.FindAllByResource("MyServer1").FirstOrDefault();
```

Ist diese vorhanden

```
                if (creds != null)
                {
```

bestimmen wir den Nutzernamen:

```
                    string username = creds.UserName;
```

Das Abrufen des Passworts gestaltet sich etwas aufwändiger:

```
                    string password = vault.Retrieve("MyServer1", username).Password;
                    ListBox1.Items.Add("Username:" + username);
                    ListBox1.Items.Add("Passwort:" + password);
                }
            }
            catch (Exception)
            {
                ListBox1.Items.Add("Keine Daten vorhanden!");
            }
...
```

Löschen der Daten:

```
...
            var vault = new PasswordVault();
```

Gelöscht wird über den Ressourcenamen und den Namen:

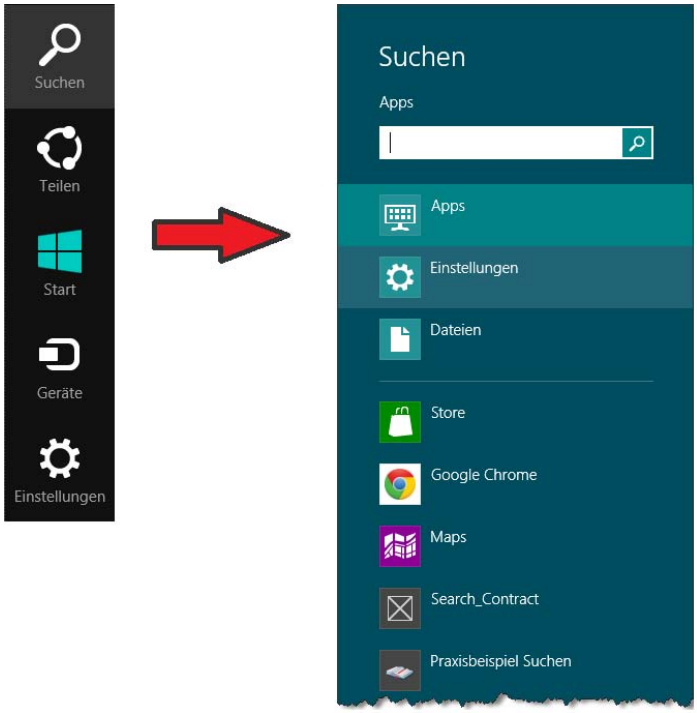
```
            vault.Remove(vault.Retrieve("MyServer1", "MayerMüller"));
...
```

Damit steht Ihnen eine einfach nutzbare Variante für das Speichern von Passwörtern zur Verfügung, die Sie beispielsweise für das Verschlüsseln der App-Daten nutzen können.

20.4 Praxisbeispiele

20.4.1 Unterstützung für den Search-Contract bieten

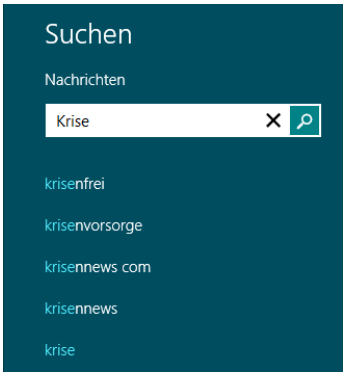
Mit dem folgenden Beispiel zeigen wir Ihnen, wie Sie einen Search-Contract in Ihrer App nutzen können. Das Grundprinzip der Suche unter Windows 8 dürfte Ihnen bereits bekannt sein, über den Charmbar (rechter Bildschirmrand, *Windows+C*) haben Sie Zugriff auf die Windows Suchfunktion, die sich in Teilen grundlegend von der bisher bekannten Suche unterscheidet:



Neben den drei Windows-Rubriken (Apps, Einstellungen, Dateien) können Sie mit dieser Suche auch in bzw. mit Hilfe anderer Apps suchen lassen. Diese Apps, die über einen entsprechenden Search-Contract verfügen, werden in der Liste unter den drei genannten Rubriken aufgelistet.

Geben Sie in das obige Suchfeld einige Zeichen ein, versucht Windows entsprechende Vorschläge (maximal 5) für mögliche Eingaben zu machen, die unter dem Suchfeld angezeigt werden. Klicken Sie auf einen dieser Einträge, wird die Suche gestartet (siehe folgende Abbildung).

HINWEIS: Einer Richtlinie von Microsoft entsprechend soll bei nur einem Fundergebnis möglichst gleich eine Detailansicht gestartet werden, nicht erst die Liste der Fundstellen.



Doch was muss ihre App nun realisieren, damit diese zum einen in der Liste der verfügbaren Apps auftaucht, zum anderen wie reagieren Sie auf Eingaben oder eine Auswahl im Suchfenster?

Folgende Voraussetzungen muss Ihre App erfüllen:

- ein aktivierter Search-Contract
- eine Callbackfunktion für die Anzeige von Vorschlägen
- eine Einstiegsmethode für die eigentliche Suche
- eine Seite für die Suchergebnisse
- optional eine Seite für die Details der Suchergebnisse

Was sich recht kompliziert anhört ist teilweise auch etwas diffizil in der Umsetzung, aus diesem Grund hat Microsoft unter den Vorlagen (*Element hinzufügen|Windows Store|Suchvertrag*) eine entsprechende Seite für die Fundergebnisse bereitgestellt. Leider wird nach einem Klick auf diese Vorlage die erste Freude schnell getrübt, ist doch die Umsetzung selten kompliziert und verworren. Aus diesem Grund stellen wir Ihnen lieber eine eigene Lösung vor, diese können Sie (sowohl was den Suchalgorithmus als auch die Darstellung betrifft) an Ihre Bedürfnisse anpassen.

Oberfläche

Erstellen Sie zunächst ein neues WinRT-Projekt und fügen Sie drei Standardseiten ein:

- eine Seite (*MainPage.xaml*) als normales Hauptformular (entspricht der regulären Programmfunktion),
- eine Seite für die Anzeige der Fundergebnisse (*SearchPage.xaml*),
- eine Seite für die Darstellung der Funddetails (*DetailPage.xaml*).

Zur Anzeige der Fundergebnisse in *SearchPage.xaml* nutzen wir eine einfache *ListBox*, der XAML-Code fällt also recht kurz aus:

```
...
        <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                      Grid.Row="1" ZoomMode="Disabled" Margin="120,0,50,50">
```

```
            <ListBox Name="ListBox1" SelectionChanged="ListBox1_SelectionChanged_1" />
        </ScrollViewer>
...
```

Auch die Seite für die Details beschränkt sich auf das Wesentliche, d h. auf einen *TextBlock* zur Anzeige des Fundes:

```
...
        <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled" Margin="120,50,50,50">
            <TextBlock HorizontalAlignment="Center" Name="txt1" FontSize="72" />
        </ScrollViewer>
...
```

HINWEIS: Öffnen Sie die Manifestdatei *Package.appxmanifest* und fügen Sie unter Deklarationen den Eintrag *Suchen* hinzu. Konfigurationen sind an dieser Stelle nicht nötig.

Quelltext (App.xaml.cs)

Damit sind wir auch schon in unserer "Schaltzentrale" angekommen. Zunächst müssen wir uns um ein paar Dummy-Datensätze kümmern, die wir im Weiteren durchsuchen lassen. Sie könnten hier auch umfangreichere Collections mit mehreren Eigenschaften zur Verfügung stellen, in unserem Beispiel genügt eine einfache Namensliste:

```
...
```

Der folgende Namespace ermöglicht uns den Zugriff auf die statische *SearchPane*-Klasse, mit der Sie das Such-Panel steuern und abfragen können:

```
using Windows.ApplicationModel.Search;
...
    sealed partial class App : Application
    {
```

Unsere "umfangreiche" Suchliste:

```
        public static readonly string[] suchliste =
            { "Anton", "Berta", "Cäsar", "Doris", "Emil", "Friedrich", "Gustav", "Heinrich",
              "Ida", "Julius", "Konrad", "Ludwig", "Marta", "Nele", "Otto", "Paula",
              "Bernhard", "Eberhard", "Eddy", "Egon", "Eike", "Edmonton", "Elmar", "Zacharias",
              "Zeno", "Pawel"
            };
```

Die Liste der späteren Fundstellen wird hier zentral für die App verwaltet, so kann diese später in der Suchseite angezeigt werden:

```
        public static IEnumerable<string> ergebnisliste;
```

Auch die Suchanfrage im Such-Panel speichern wir für die Anzeige:

```
        public static string suchanfrage = "";
```

Das finale Suchergebnis (ein Eintrag) für die Anzeige in der Detailseite:

```
        public static string ergebnis = "";

        public App()
        {
            this.InitializeComponent();
            this.Suspending += OnSuspending;
        }
```

Wird das Programm regulär (*OnLaunched*) gestartet, klinken wir uns in das Such-Panel ein und stellen per Ereignis eine Liste von Suchvorschlägen zur Verfügung:

```
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
...
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
            SearchPane.GetForCurrentView().SuggestionsRequested += OnSuchvorschlägeFiltern;
        }
```

Der eigentliche Eventhandler:

```
        void OnSuchvorschlägeFiltern(SearchPane sender,
                                     SearchPaneSuggestionsRequestedEventArgs args)
        {
```

Per Parameter wird der bisher eingegebene Suchtext überreicht:

```
            var suchtext = args.QueryText.ToUpper();
```

Mittels *Request* können wir eine Liste von Suchvorschlägen an das Such-Panel zurückgeben. Dazu filtern wir unsere Suchliste nach den gewünschten Einträgen (per LINQ-Abfrage) und beschränken die Anzahl der Vorschläge auf fünf (mehr werden auch nicht angezeigt):

```
            var request = args.Request;
            if (!string.IsNullOrEmpty(suchtext))
            {
                request.SearchSuggestionCollection.AppendQuerySuggestions(
                    suchliste.Where(s => s.ToUpper().StartsWith(suchtext)).Take(5));
            }
        }
```

Wie Sie die Stringliste an die *SearchSuggestionCollection* übergeben hängt von den Gegebenheiten Ihres Programms ab. Beachten Sie jedoch, dass diese Aktion nicht zu lange dauert, Sie wissen ja: ... fast and fluid ...

HINWEIS: Zum jetzigen Zeitpunkt wurde nur auf die Eingabe eines Wertes im Such-Panel reagiert und eine Liste von Vorschlägen im Panel angezeigt.

Kommen wir nun zu der Aktion, in der die eigentliche Suche "angestoßen" wird, d h., der Nutzer bestätigt die Eingabe im Suchfeld oder er wählt einen Eintrag aus der Vorschlagsliste aus. In diesem Fall wird die Methode *OnSearchActivated* aufgerufen.

```
protected override void OnSearchActivated(
                Windows.ApplicationModel.Activation.SearchActivatedEventArgs args)
{
```

Beachten Sie, dass es zwei Gründe geben kann, warum diese Methode gestartet wird.

Variante1: Die App läuft bereits und der Nutzer hat einen Suchtext eingegeben und bestätigt die Eingabe, oder er nutzt die Vorschlagliste.

Variante 2: Die App wird direkt durch Auswahl der App im Such-Panel aktiviert, und wir haben es an dieser Stelle mit dem Start der App zu tun. Auch hier müssen wir eine Liste von Vorschlägen bereitstellen.

Wir versuchen also zunächst eine bestehende Anzeigekonfiguration (Window→Frame) abzurufen, ist diese noch nicht vorhanden, erstellen wir einen neuen Frame (Erststart der App):

```
var previousContent = Window.Current.Content;
var frame = previousContent as Frame;
if (frame == null)
{
    frame = new Frame();
    Window.Current.Content = frame;
    SearchPane.GetForCurrentView().SuggestionsRequested += OnSuchvorschlägeFiltern;
}
```

Wir rufen den Eingabetext ab:

```
suchanfrage = args.QueryText.ToUpper();
```

Handelt es sich um eine Leereingabe, so erstellen wir auch eine leere Fundliste, alternativ könnten Sie hier auch alle Datensätze zur Anzeige auswählen:

```
if (string.IsNullOrEmpty(suchanfrage))
    ergebnisliste = Enumerable.Empty<string>();
else
```

Bei korrekter Suchanfrage filtern wir unsere Daten und stellen diese in der Ergebnisliste bereit:

```
    ergebnisliste = App.suchliste.Where(s => s.ToUpper().StartsWith(suchanfrage));
```

Die Anzeige der jeweiligen Seite hängt nun von der Anzahl der Fundergebnisse ab. Normalerweise erscheint jetzt die Seite mit den Fundergebnissen:

```
Type zielpage = typeof(SearchPage);
```

Wird nur ein Eintrag gefunden, können wir gleich die Detailseite anzeigen und legen auch hier schon das Fundergebnis fest. Alternativ könnten Sie hier Daten kopieren oder anderweitig bereitstellen und lediglich eine Meldung über eine erfolgreiche Suche anzeigen:

```
if (ergebnisliste.Count() == 1)
{
```

```
                ergebnis = ergebnisliste.First();
                zielpage = typeof(DetailPage);
            }
```

Die Navigation starten und das Fenster aktivieren:

```
            frame.Navigate(zielpage);
            Window.Current.Activate();
        }
    }
```

Damit ist unsere "Schaltzentrale" fertig.

Quelltext (SearchPage.xaml.cs)

Wird die Suchseite geöffnet, nutzen wir die Ergebnisliste zunächst zur Beschriftung und nachfolgend zur Datenbindung für die Anzeige der Fundstellen:

```
...
   public sealed partial class SearchPage : Search_Beispiel.Common.LayoutAwarePage
    {
...
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            if (App.ergebnisliste.Count() > 0)
                pageTitle.Text = "Suchergebnisse für '" + App.suchanfrage + "':";
            else
                pageTitle.Text = "Keine Suchergebnisse für '" + App.suchanfrage +
                                 "' vorhanden!";
            ListBox1.ItemsSource = App.ergebnisliste;
            base.OnNavigatedTo(e);
        }
```

Wird ein Eintrag in der Fundliste ausgewählt, haben wir nur noch ein finales Ergebnis, das wir auf der Detailseite anzeigen können (auch hier sind andere Aktionen möglich):

```
        private void ListBox1_SelectionChanged_1(object sender, SelectionChangedEventArgs e)
        {
            string auswahl = (ListBox1.SelectedItem as string);
            App.ergebnis = auswahl;
            this.Frame.Navigate(typeof(DetailPage));
        }
    }
```

Quelltext (DetailPage.xaml.cs)

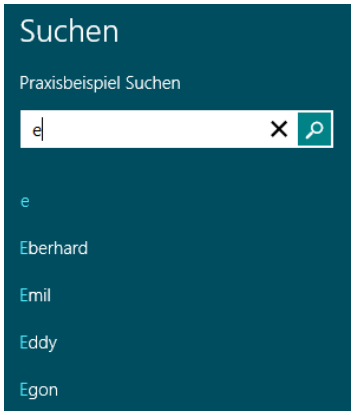
Hier beschränken wir uns beispielhaft auf die einfache Anzeige des final gewählten Eintrags der Suche:

```
...
    public sealed partial class DetailPage : Search_Beispiel.Common.LayoutAwarePage
    {
```

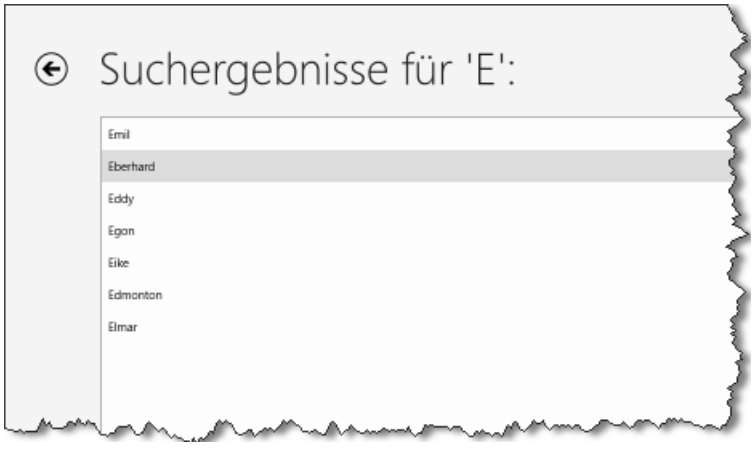
```
...
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            txt1.Text = App.ergebnis;
            base.OnNavigatedTo(e);
        }
    }
```

Test

Erstellen Sie das Projekt und starten Sie dieses. Rufen Sie jetzt über den Charmbar die Suche auf. In der Liste der potenziellen Such-Apps sollte auch Ihre Anwendung auftauchen und gegebenenfalls bereits markiert sein. Geben Sie in das Suchfeld den Buchstaben "E" ein, sollten von der App einige Vorschläge im Panel angezeigt werden:



Ein Klick auf einen der Beispieleinträge lässt die App zur Detailseite springen, andernfalls erscheint eine Liste mit den Fundstellen:



Erweiterungen

Möchten Sie aus Ihrer App heraus direkt das Such-Panel anzeigen, ist auch dies kein Problem. Nutzen Sie dazu einfach die folgende Methode:

```
...
using Windows.ApplicationModel.Search;
...
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            SearchPane.GetForCurrentView().Show("E");
        }
```

HINWEIS: Sie können, müssen aber nicht, einen Suchtext übergeben.

20.4.2 Die Auto-Play-Funktion unterstützen

Als Beispiel für eine Extension möchten wir Ihnen im Folgenden zeigen, wie Sie mit Ihrer App auf das Einstecken einer SD-Card mit Bildern einer Digitalkamera reagieren können.

HINWEIS: An dieser Stelle geht es uns nicht um einen neuen Bildbetrachter, sondern um das Implementieren einer Extension, die auf externe Ereignisse reagiert.

Welche Funktionen müssen wir realisieren?

- Der Anwendung den Zugriff auf das Wechselmedium gestatten.
- Deklarationen für die unterstützten Dateitypen (.JPG, .JPEG) hinzufügen.
- Eine Deklaration für die automatische Wiedergabe hinzufügen.
- Die Methode *OnFileActivated* implementieren.

Etwas Hintergrundwissen zu diesem Thema ist unabdingbar: Die Bilder auf der SD-Card einer Digitalkamera werden nach folgendem Muster gespeichert:

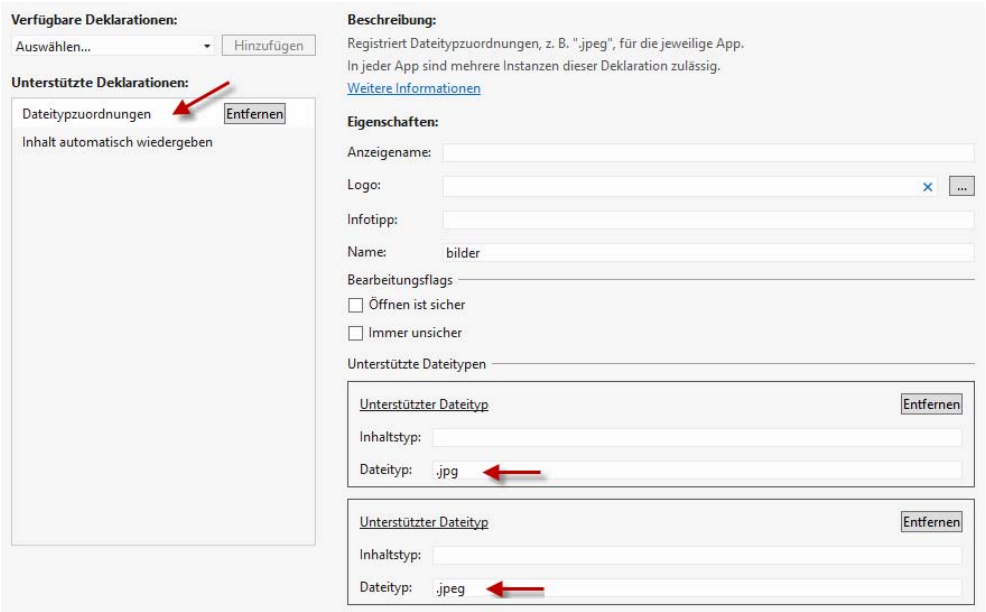
```
\\DCIM\<Liste von Unterordnern>\<Liste der Bilder>
```

Wollen wir auf die Liste der Bilder zugreifen, müssen wir die obigen Unterverzeichnisse komplett durchsuchen, andernfalls wird unser Programm keine Bilder finden.

Anpassen Package.appxmanifest

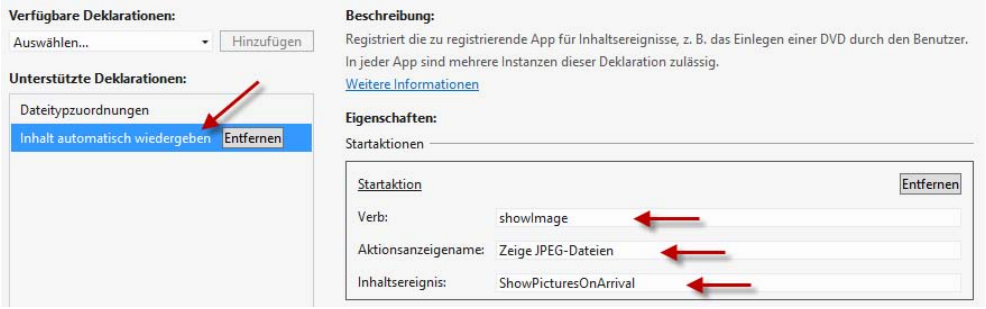
In der Rubrik *Funktionen* aktivieren Sie die Option "Wechselspeichergeräte", um Zugriff auf diese zu erhalten.

In der Rubrik *Deklarationen* fügen Sie zunächst eine *Datentypzuordnung* hinzu und ergänzen diese, so wie es in der Abbildung zu sehen ist:



Nächster Schritt ist das Hinzufügen einer Deklaration für "Inhalt automatisch wiedergeben". Hier müssen Sie

- ein Verb festlegen (z.B. *showImage*),
- einen Anzeigenamen definieren (dieser wird beim ersten Verknüpfen Ihrer App mit dem Autoplay-Ereignis angezeigt),
- das gewünschte Ereignis angeben (*ShowPicturesOnArrival*).



Achten Sie beim Inhaltsereignis unbedingt auf die korrekte Schreibweise. Welche weiteren Ereignisse (*WPD\ImageSource*, *WPD\VideoSource*, *StorageOnArrival*, *PlayMusicFilesOnArrival* ...) noch unterstützt werden, können Sie der Online-Hilfe entnehmen.

Quelltext

Damit haben wir schon drei der vier Voraussetzungen erfüllt, damit unsere App auch vom System als potenzielle Auto-Play-App erkannt wird. Was noch fehlt ist die eigentliche Reaktion auf das Einlegen der SD-Card, d h., wir müssen noch die Methode *OnFileActivated* implementieren. Dazu nutzen wir die bereits vorhandene Klasse *App* und erweitern diese um die folgende Methode:

```
async protected override void OnFileActivated(FileActivatedEventArgs args)
{
```

Da diese Methode die Funktion der *OnLauched*-Methode komplett ersetzt, wenn die App per AutoPlay gestartet wird, müssen wir uns auch hier zunächst um das Erstellen der Startseite und die Zuweisung zum App-Window kümmern[1]:

```
    Frame rootFrame = null;
    if (Window.Current.Content == null)
    {
        rootFrame = new Frame();
        rootFrame.Navigate(typeof(BasicPage1));
        Window.Current.Content = rootFrame;
    }
    else
    {
        rootFrame = Window.Current.Content as Frame;
    }
    Window.Current.Activate();
```

Ab hier kümmern wir uns dann um die Abfrage des neu eingesteckten Wechseldatenträgers:

Instanz der sichtbaren Seite abrufen (über die darin enthaltene *ListBox* zeigen wir den Inhalt des Wechseldatenträgers an):

```
    BasicPage1 page1 = (BasicPage1)rootFrame.Content as BasicPage1;
```

Über den Parameter *args.Verb* können Sie unterscheiden aus welchem Grund die App aufgerufen wurde (diesen Bezeichner haben wir in der Deklaration vorgegeben):

```
    if (args.Verb == "showImage")
    {
        page1.ListBox1.Items.Clear();
        page1.ListBox1.Items.Add("args.verb = showImage");
```

Wir rufen alle übergebenen Verzeichnisse ab (es ist in diesem Fall nur eins):

```
        foreach (var file in args.Files)
        {
            page1.ListBox1.Items.Add("Laufwerk: " + file.Name);
```

Wir ermitteln den Ordner *DCIM* und alle untergeordneten Verzeichnisse:

```
            StorageFolder verz = file as StorageFolder;
            var verz_dcim = await verz.GetFolderAsync("DCIM");
```

[1] Starten Sie die App per Windows 8-Desktop, wird jedoch *OnLaunched* ausgeführt.

```
                foreach (StorageFolder subverz in await verz_dcim.GetFoldersAsync())
                {
```

Alle enthaltenen Dateien (.JPG und .JPEG) auflisten:

```
                    foreach (StorageFile f in await subverz.GetFilesAsync())
                    {
                        page1.ListBox1.Items.Add(f.Path);
                    }
                }
            }
        }
    }
```

HINWEIS: Beachten Sie, dass Ihnen nur die unterstützten Dateitypen angezeigt werden. Eventuell enthaltene .AVI-Dateien tauchen später nicht in der Liste auf, da wir diesen Dateityp nicht "angemeldet" haben.

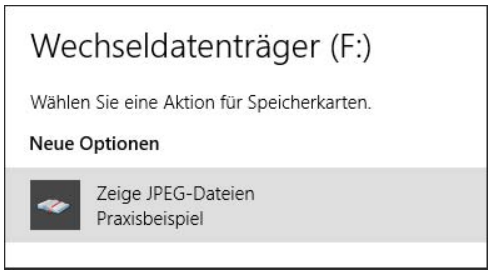
Oberfläche

Lediglich eine Seite mit einer enthaltenen *ListBox*. Der Zugriff auf diese muss per *FieldModifier* freigegeben werden, sonst können wir aus der Klasse *App* nicht darauf zugreifen.

```
...
    <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled" Margin="120,0,0,0">
        <ListBox Name="ListBox1"  x:FieldModifier="public" />
    </ScrollViewer>
...
```

Test

Starten Sie die App einmal und beenden Sie diese wieder. Stecken Sie jetzt eine SDCard aus einer Digitalkamera in das entsprechende Laufwerk. Zunächst sollte in der rechten oberen Ecke des Desktops folgende Meldung erscheinen:



Die Beschriftung und das Symbol für die neue Aktion entspricht den Einstellungen in Ihrer App. Wählen Sie den neuen Eintrag und nach kurzer Zeit erscheint die App mit der Liste der Bilddateien (intern ruft WinRT jetzt die Methode *OnFileActivated* mit dem Verb *showImage* auf):

HINWEIS: Stecken Sie später erneut eine SD-Card ins Laufwerk, erscheint die App ohne vorherige Abfrage, die Verknüpfung ist ja bereits vorhanden.

20.4.3 Einen zusätzlichen Splash Screen einsetzen

Leider ist nicht jedes Programm so schnell wie wir es gern hätten, und so kommen wir in einigen Fällen mit den maximal 15 Sekunden für die Anzeige der ersten App-Seite nicht immer aus. Bevor Sie jetzt verzweifelt nach einer wie auch immer gearteten Lösung suchen, sollten Sie sich folgende Variante näher ansehen:

Wir erstellen eine dem Standard-Splash Screen entsprechende Seite, erweitern diese um eine Fortschrittsanzeige und zeigen dieses Formular als erstes Formular der App an. Ist der Ladevorgang (wir simulieren diesen per Pause) abgeschlossen, wechseln wir zum eigentlichen Hauptformular der App. Warum das gleiche Aussehen? Auf diese Weise vermeiden wir den Eindruck, dass es zwei Splash Screens gibt, was leider der Fall ist, da der Standard-Splash Screen nicht deaktivierbar ist.

Oberfläche (ErsatzSplashscreen.xaml)

Erstellen Sie zunächst ein neues WinRT-App-Projekt und fügen Sie eine leere Seite unter dem Namen *ErsatzSplashscreen.xaml* ein. Erweitern Sie diese wie folgt:

```
...
   <Grid Background="#FF0420A0" >
      <ProgressBar Name="Progressbar1" Maximum="10" Width="500" Height="50"
                   Margin="0,350,0,0" />
        <Image Source="Assets/SplashScreen.png" Stretch="None" />
    </Grid>
...
```

HINWEIS: Die Abbildung entspricht dem Symbol des eigentlichen Splash Screens.

Quellcode (App.xaml.cs)

Wir navigieren zum Ersatz-Splash Screen und übergeben das *SplashScreen*-Objekt (das ist der aktuell sichtbare Original-Splash Screen) als Parameter:

```
...
       async protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
...
            var rootFrame = new Frame();
            if (!rootFrame.Navigate(typeof(ErsatzSplashScreen), args.SplashScreen))
            {
                throw new Exception("Failed to create initial page");
            }

            // Place the frame in the current Window and ensure that it is active
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
```

Quellcode (ErsatzSplashScreen.xaml.cs)

Fügen Sie zunächst folgenden Namespace hinzu:

```
...
using Windows.ApplicationModel.Activation;
...
    public sealed partial class ErsatzSplashScreen : Page
    {
```

Wird die Seite geöffnet, nutzen wir das per Parameter übergebene *SplashScreen*-Objekt, um per *Dismissed*-Ereignis auf das Schließen des originalen Splash Screens zu reagieren:

```
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            SplashScreen splash = (SplashScreen)(e.Parameter);
            if (splash != null)
                splash.Dismissed += splash_Dismissed;
        }
```

Tritt das Ereignis ein, könne wir die Fortschrittsanzeige in Gang setzen (eventuell laden Sie bereits vorher asynchron Daten, die Sie benötigen):

```
        async void splash_Dismissed(SplashScreen sender, object args)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                DatenLaden();
            });
```

```
        }
```

Die Anzeige des Fortschritts:

```
        async public void DatenLaden()
        {
            for (int i = 0; i <= 10; i++)
            {
                await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Low, () =>
                {
                    Progressbar1.Value = i;
                });
```

Hier simulieren wir Aufgaben, die von der App zu erledigen sind:

```
                Task.Delay(1000).Wait();
            }
```

Sind alle Daten geladen, kann das eigentliche Hauptformular gestartet werden:

```
            this.Frame.Navigate(typeof(MainPage));
        }
...
```

Test

Starten Sie die App, so sollte zunächst der Standard-Splash-Screen erscheinen, nachfolgend flackert eventuell die Anzeige und es erscheint unser Splash Screen mit der Fortschrittsanzeige. Ist diese abgelaufen, öffnet sich die eigentliche Hauptseite.

20.4.4 Eine Dateiverknüpfung erstellen

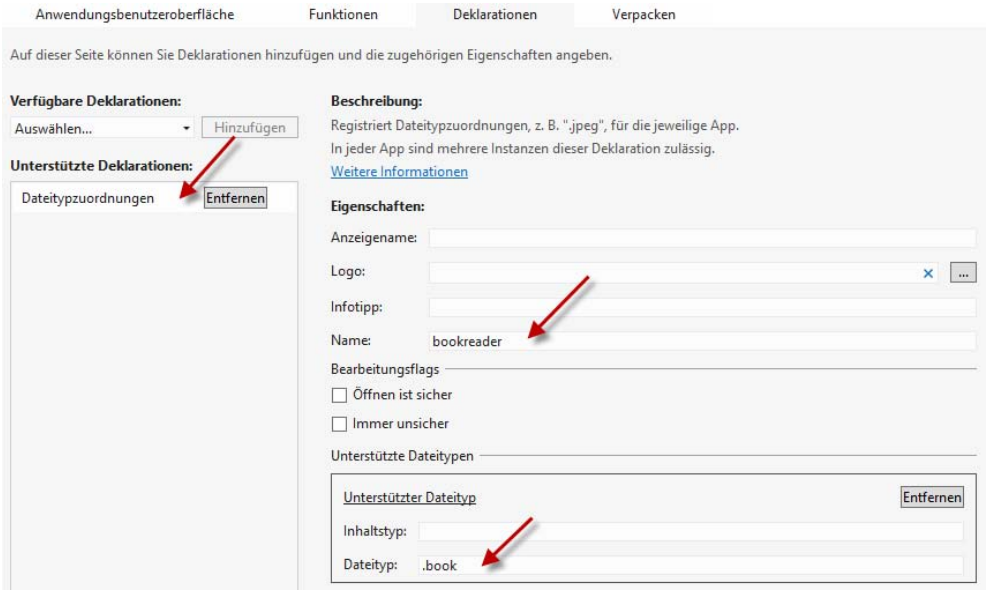
Die Grundfunktionalität einer Dateiverknüpfung dürfte Ihnen sicher bekannt sein: Der Anwender klickt auf eine Datei, und es öffnet sich die entsprechende Anwendung zur Bearbeitung. Gleiches wollen wir auch in einer App realisieren. Dazu erstellen wir eine *Dateitypzuordnung* zum fiktiven Dateityp *.book* (umbenannte Textdateien). Aus Entwicklersicht handelt es sich um eine *File Type Association*-Extension, die wir entsprechend definieren/implementieren müssen.

Oberfläche

Erstellen Sie lediglich eine Seite mit einer *ListBox*:

```
...
        <ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
                  Grid.Row="1" ZoomMode="Disabled" Margin="120,50,50,50">
            <ListBox Name="ListBox1"/>
        </ScrollViewer>
...
```

Zusätzlich müssen Sie der Manifestdatei *Package.appxmanifest* noch eine *Dateitypzuordnung* hinzufügen, nur so erlangen Sie Zugriff auf die Datei, und es wird eine Verknüpfung zwischen App und Datei hergestellt:



HINWEIS: Sie können Ihre App auch für mehrere Dateitypen registrieren.

Quelltext (App.xaml.cs)

Wird eine App durch den Klick auf verknüpfte Dateien geöffnet, ändert sich der Eintrittspunkt in die App. Statt *OnLauched* müssen Sie jetzt die Methode *OnFileActivated* implementieren. Mit dem übergebenen Parameter *args.Files* lassen sich die Dateinamen auswerten.

Erweitern Sie also die Klasse *App* um folgende Zeilen:

```
...
using Windows.ApplicationModel.Activation;
...
        protected override void OnFileActivated(FileActivatedEventArgs args)
        {
```

Der Ablauf entspricht dem bei einem normalen Programmstart:

```
            if (args.PreviousExecutionState == ApplicationExecutionState.Running)
            {
                Window.Current.Activate();
                return;
            }
            var rootFrame = new Frame();
```

Wir übergeben jedoch zusätzlich die Aufrufparameter an die anzuzeigende Seite:

```
            rootFrame.Navigate(typeof(BasicPage1), args );
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
            base.OnFileActivated(args);
        }
```

Quelltext (BasicPage1.xaml.cs)

Wir die Seite geöffnet, werten wir die Parameter (d h. die Dateiliste) aus:

```
...
using Windows.ApplicationModel.Activation;
using Windows.Storage;
...
        async protected override void OnNavigatedTo(NavigationEventArgs e)
        {
```

Nur wenn die Seite auch mit Parametern gestartet wird können wir etwas auslesen:

```
            if (e.Parameter != null)
            {
```

File-Liste ermitteln:

```
                FileActivatedEventArgs args = (FileActivatedEventArgs)e.Parameter;
```

Jeden Eintrag auswerten und den Dateiinhalt anzeigen:

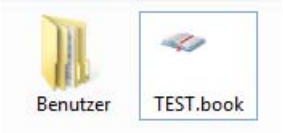
```
                foreach (var file in args.Files)
                {
                    ListBox1.Items.Add("Bearbeiten: " + file.Path);
                    string inhalt = await FileIO.ReadTextAsync((StorageFile)file);
                    ListBox1.Items.Add(inhalt);
                }
            }
        }
```

HINWEIS: Achten Sie auf die asynchrone Ausführung!

Test

Starten Sie das Programm, um es zu installieren, und beenden Sie es gleich wieder. Windows hat zu diesem Zeitpunkt bereits die Manifestdatei ausgewertet und damit auch die Dateiverknüpfung zum Typ *.book* im System registriert.

Erstellen Sie auf dem Desktop eine Textdatei und benennen Sie diese in *Test.book* um. Nachfolgend sollte sich auch das Symbol der Datei anpassen, es erscheint das Icon unserer App:



Klicken Sie mit der rechten Maustaste auf die Datei, um im Menü den Eintrag *Öffnen mit ...* zu wählen. Der folgende Dialog zeigt Ihnen, dass die Verknüpfung bereits ordnungsgemäß registriert ist:

Ein Doppelklick auf die obige Datei bringt unsere App zum Vorschein, in der Liste sollte der Name und der Inhalt der Datei angezeigt werden:

Sie können natürlich auch mehrere Dateien erstellen, alle markieren und dann über das Kontextmenü den Eintrag *Öffnen* wählen. In diesem Fall wird nur eine Instanz der App gestartet, diese erhält eine Liste der zu bearbeitenden Dateien.

Ergänzungen

Möchten Sie diese App debuggen, so setzen Sie einfach einen Breakpoint in die *OnFileActivated*-Methode. Starten Sie jetzt noch nicht den Debugger, sondern wählen Sie unter *Projekt|Eigenschaften|Debuggen* die Option *Eigenen Code zunächst nicht starten sondern debuggen*.

Starten Sie erst jetzt den Debugger, es sollte zunächst nichts passieren. Klicken Sie jetzt auf eine der *.book*-Dateien um den Methodenaufruf bzw. den App-Aufruf auszulösen. Der Debugger stoppt jetzt an der gewünschten Stelle.

Aus einer anderen WinRT-App können Sie die vorliegende App mit folgendem Aufruf starten:

```
async private void Button_Click_1(object sender, RoutedEventArgs e)
 {
     StorageFile file = await
                          KnownFolders.DocumentsLibrary.GetFileAsync("Buchtest.book");
     await Launcher.LaunchFileAsync(file);
 }
```

Sie rufen also nicht die App direkt auf, sondern öffnen quasi die betreffende Datei. Windows stellt Ihnen dann die zugeordnete App zur Verfügung und öffnet diese entsprechend.

Kapitel 21

WinRT-Techniken

Dieses Kapitel versteht sich nicht als eine Wiederholung der entsprechenden Microsoft-Dokumentation, sondern als Hilfe für die Lösung spezifischer Programmieraufgaben im Umfeld einer WinRT-App. Aus diesem Grund kann es auch vorkommen, dass sich einige Erläuterungen wiederholen, dies ist jedoch sinnvoller als ein dauerndes Verweisen auf andere Abschnitte dieses oder anderer Kapitel.

HINWEIS: Die Themen dieses Kapitels stehen relativ lose nebeneinander, Sie können also problemlos auch einen oder mehrere Abschnitte überspringen.

21.1 Arbeiten mit Dateien/Verzeichnissen

Nachdem wir im vorhergehenden Kapitel bereits die Grundlagen des Datenhandlings vorgestellt haben, soll es jetzt etwas spezieller werden. Wir wollen uns die grundlegenden Datei- und Verzeichnisoperationen näher ansehen, da diese Basis für das effiziente Ablegen der Daten sind.

HINWEIS: Wie Sie zunächst Zugriff auf diverse Verzeichnisse erhalten und welche Verzeichnisse Sie überhaupt nutzen können, hatten wir bereits im Abschnitt 20.3 ab Seite1018 besprochen.

21.1.1 Verzeichnisinformationen auflisten

Haben Sie es geschafft ein *StorageFolder*-Objekt zu initialisieren (Abschnitt 20.3), können Sie dieses nutzen, um mehr über den Ordner zu erfahren.

BEISPIEL 21.1: Abrufen von Informationen über ein *StorageFolder*-Objekt

C#
```
using Windows.Storage;
...
        async private void Button_Click_2(object sender, RoutedEventArgs e)
        {
```

BEISPIEL 21.1: Abrufen von Informationen über ein *StorageFolder*-Objekt

Zunächst ermitteln wir einige Trivialeigenschaften des Folders. Dabei können wir direkt auf die Eigenschaften des *StorageFolder*-Objekts zugreifen.

Wir verwenden den App-Datenspeicher:

```
StorageFolder folder = ApplicationData.Current.LocalFolder;
ListBox1.Items.Add("Name: " + folder.Name);
ListBox1.Items.Add("Path: " + folder.Path);
ListBox1.Items.Add("Attributes: " + folder.Attributes.ToString());
ListBox1.Items.Add("DateCreated: " + folder.DateCreated.ToString());
ListBox1.Items.Add("DisplayName: " + folder.DisplayName);
ListBox1.Items.Add("DisplayType: " + folder.DisplayType);
ListBox1.Items.Add("FolderRelativeID: " + folder.FolderRelativeId);
```

Geht es um etwas speziellere Eigenschaften, müssen wir uns schon die Mühe machen, diese asynchron abzurufen:

```
var prop = await folder.GetBasicPropertiesAsync();
ListBox1.Items.Add("Properties.DateModified: " + prop.DateModified);
ListBox1.Items.Add("Properties.ItemDate: " + prop.ItemDate);
ListBox1.Items.Add("Properties.Size: " + prop.Size);
```

Nicht genug der Pein, es wartet noch eine Eigenschaftenauflistung auf Sie:

```
var props = await prop.RetrievePropertiesAsync(new List<string> { });
foreach (var p in props)
{
  if (p.Value != null)
       ListBox1.Items.Add("RetrievePropertiesAsync: " +
                          p.Key + ": " + p.Value.ToString());
}
      }
...
```

Ergebnis

Das Ergebnis unserer Bemühungen:

```
Name: LocalState
Path: C:\Users\Thomas\AppData\Local\Packages\1fc01a7c-1433-4074-ab86-
517c59b8d321_xkj04xcm6e7n4\LocalState
Attributes: Directory
DateCreated: 01.01.1601 01:00:00 +01:00
DisplayName: LocalState
DisplayType: Dateiordner
FolderRelativeID: 0\0\LocalState
Properties.DateModified: 14.08.2012 11:12:12 +02:00
Properties.ItemDate: 14.08.2012 11:12:12 +02:00
Properties.Size: 0
RetrievePropertiesAsync: System.KindText: Ordner
RetrievePropertiesAsync: System.ZoneIdentifier: 3
RetrievePropertiesAsync: System.Shell.SFGAOFlagsStrings: System.String[]
```

BEISPIEL 21.1: Abrufen von Informationen über ein *StorageFolder*-Objekt

```
RetrievePropertiesAsync: System.ItemPathDisplayNarrow: LocalState
(C:\Users\Thomas\AppData\Local\Packages\1fc01a7c-1433-4074-ab86-517c59b8d321_xkj04xcm6e7n4)
RetrievePropertiesAsync: System.ItemName: LocalState
RetrievePropertiesAsync: System.PerceivedType: 2
RetrievePropertiesAsync: System.ParsingName: LocalState
RetrievePropertiesAsync: System.Kind: System.String[]
RetrievePropertiesAsync: System.ItemFolderNameDisplay: 1fc01a7c-1433-4074-ab86-
517c59b8d321_xkj04xcm6e7n4
RetrievePropertiesAsync: System.ParsingPath:
C:\Users\Thomas\AppData\Local\Packages\1fc01a7c-1433-4074-ab86-
517c59b8d321_xkj04xcm6e7n4\LocalState
RetrievePropertiesAsync: System.VolumeId: cd01e3d7-d334-497a-a90e-7f27ee556bbf
RetrievePropertiesAsync: System.ItemTypeText: Dateiordner
RetrievePropertiesAsync: System.ItemFolderPathDisplayNarrow: 1fc01a7c-1433-4074-ab86-
517c59b8d321_xkj04xcm6e7n4 (C:\Users\Thomas\AppData\Local\Packages)
RetrievePropertiesAsync: System.FileName: LocalState
RetrievePropertiesAsync: System.SharedWith: System.String[]
RetrievePropertiesAsync: System.ItemPathDisplay:
C:\Users\Thomas\AppData\Local\Packages\1fc01a7c-1433-4074-ab86-
517c59b8d321_xkj04xcm6e7n4\LocalState
RetrievePropertiesAsync: System.SFGAOFlags: 1887437183
RetrievePropertiesAsync: System.ItemType: Directory
RetrievePropertiesAsync: System.FileAttributes: 16
RetrievePropertiesAsync: System.ComputerName: WINDOWS8RC
RetrievePropertiesAsync: System.ItemNameDisplay: LocalState
RetrievePropertiesAsync: System.FileOwner: WINDOWS8RC\Thomas
RetrievePropertiesAsync: System.SharingStatus: 1
RetrievePropertiesAsync: System.IsShared: True
RetrievePropertiesAsync: System.ItemNameDisplayWithoutExtension: LocalState
RetrievePropertiesAsync: System.ItemFolderPathDisplay:
C:\Users\Thomas\AppData\Local\Packages\1fc01a7c-1433-4074-ab86-517c59b8d321_xkj04xcm6e7n4
RetrievePropertiesAsync: System.NotUserContent: False
...
```

HINWEIS: Optional können Sie der Methode *RetrievePropertiesAsync* eine Liste der gewünschten Eigenschaften übergeben, Sie erhalten dann auch nur diese.

Die möglichen Member der *FileAttributes*-Enumeration:

- *Normal*
- *ReadOnly*
- *Directory*
- *Archive*
- *Temporary*

21.1.2 Unterverzeichnisse auflisten

Nichts leichter als das, rufen Sie die enthaltenen Ordner per *GetFoldersAsync*-Methode ab:

BEISPIEL 21.2: Unterverzeichnisse auflisten

C#
```
...
using Windows.Storage;
...
        async private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            IReadOnlyList<StorageFolder> folders = await
                    ApplicationData.Current.LocalFolder.GetFoldersAsync();
            foreach (var folder in folders)
            {
                ListBox1.Items.Add(folder.Path);
            }
        }
...
```

So weit so gut, doch was passiert, wenn Sie vielleicht auf die verwegene Idee kommen, Verzeichnisse rekursiv zu durchsuchen?

Bevor Sie jetzt mühsam eine eigene Lösung basteln, die vermutlich erst einmal an der asynchronen Ausführung scheitert, versuchen Sie es doch mal mit den *QueryOptions* bzw. mit der Methode *CreateFolderQueryWithOptions*. Diese bietet unter anderem auch die Möglichkeit, einen Dateifilter zu implementieren.

BEISPIEL 21.3: Rekursive Suche nach Verzeichnissen

C#
```
...
using Windows.Storage;
using Windows.Storage.Search;
...
        async private void Button_Click_3(object sender, RoutedEventArgs e)
        {
```

Eine *QueryOptions*-Instanz erzeugen:

```
            QueryOptions queryOptions = new QueryOptions(CommonFolderQuery.DefaultQuery);
```

Wir wollen das Verzeichnis rekursiv durchsuchen:

```
            queryOptions.FolderDepth = FolderDepth.Deep;
```

Indexer gegebenenfalls verwenden:

```
            queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable;
```

Wir suchen nur Verzeichnisse die mit "Probe" beginnen:

```
            queryOptions.ApplicationSearchFilter = "probe*";
```

BEISPIEL 21.3: Rekursive Suche nach Verzeichnissen

Die Abfrage erstellen (bis jetzt passiert im Dateisystem noch gar nichts):

```
var query =
  ApplicationData.Current.LocalFolder.CreateFolderQueryWithOptions(queryOptions);
```

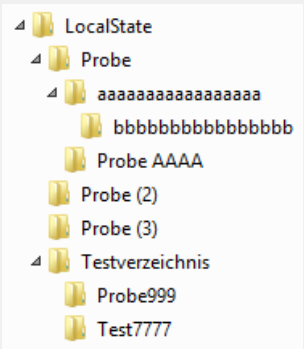
Jetzt das Verzeichnis abrufen:

```
foreach (var f in await query.GetFoldersAsync())
{
    ListBox1.Items.Add(f.Path);
}
```

...

Ergebnis

Bei folgender Verzeichnisstruktur:



erzielen wir diese Ausgabe:

```
C:\Users\...\LocalState\Probe
C:\Users\...\LocalState\Probe (2)
C:\Users\...\LocalState\Probe (3)
C:\Users\...\LocalState\Probe\Probe AAAA
C:\Users\...\LocalState\Testverzeichnis\Probe999
```

HINWEIS: Auf alle Varianten der Verzeichnissuche können wir an dieser Stelle nicht eingehen, wir empfehlen eine eingehende Beschäftigung mit den *QueryOptions*. Insbesondere die Möglichkeit, Verzeichnisse auf Änderungen zu überwachen, ist recht interessant.

Mehr dazu im Praxisbeispiel

► 21.7.1 Ein Verzeichnis auf Änderungen überwachen

21.1.3 Verzeichnisse erstellen/löschen

Sollen die Dateien im App-Datenverzeichnis oder im *Downloads*-Verzeichnis sinnvoll gruppiert werden, kommen Sie um die Verwendung von Unterverzeichnissen nicht herum.

BEISPIEL 21.4: Einige Unterverzeichnisse im App-Datenverzeichnis erstellen

C#

```
...
   await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                 CreationCollisionOption.GenerateUniqueName);
   await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                 CreationCollisionOption.GenerateUniqueName);
...
```

Die Aufrufe erzeugen (unabhängig davon ob schon ein *Probe*-Verzeichnis vorhanden ist) weitere *Probe*-Verzeichnisse mit eindeutiger Nummerierung.

Folgende Optionen (*CreationCollisionOption*) stehen beim Erzeugen des Verzeichnisses zur Verfügung:

Member	Beschreibung
GenerateUniqueName	Wie im obigen Beispiel wird im Zweifel eine Ziffer an den Dateinamen angehängt, um einen eindeutigen Verzeichnisnamen zu erstellen.
ReplaceExisting	Erzeugt eine neues Verzeichnis, der Vorgänger wird samt Inhalt ersetzt[1].
FailIfExists	Es wird ein Fehler ausgelöst, wenn die Pfad schon existiert!
OpenIfExists	Wenn vorhanden, wird das Verzeichnis geöffnet, andernfalls zunächst erstellt.

Das Löschen eines Verzeichnisses realisieren wir mit der Methode *DeleteAsync*:

BEISPIEL 21.5: Verzeichnis erstellen und löschen

C#

```
...
       async private void Button_Click_4(object sender, RoutedEventArgs e)
       {
```

Erzeugen:

```
           await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                                 CreationCollisionOption.ReplaceExisting);
```

Auswahl und Löschen:

```
           StorageFolder folder = await
                  ApplicationData.Current.LocalFolder.GetFolderAsync("Probe");
           await folder.DeleteAsync(StorageDeleteOption.PermanentDelete);
       }
...
```

[1] Diese Variante funktionierte bei den Autoren nicht fehlerfrei.

HINWEIS: Über den Parameter *StorageDeleteOption* bestimmen Sie, ob das Verzeichnis endgültig (*PermanentDelete*) oder per Papierkorb (*Default*) gelöscht wird.

21.1.4 Dateien auflisten

Die grundsätzliche Vorgehensweise entspricht der im Abschnitt 21.1.2, das folgende Beispiel zeigt drei Varianten:

BEISPIEL 21.6: Dateien auflisten

C#

```
...
using Windows.Storage;
using Windows.Storage.Search;
...
```

Die einfachste Variante, alle Dateien eines Ordners zu ermitteln:

```
IReadOnlyList<StorageFile> files = await
             ApplicationData.Current.LocalFolder.GetFilesAsync() ;
foreach (var file in files)
    ListBox1.Items.Add(file.Path);
```

Um nach Dateien rekursiv (mit Unterverzeichnissen) suchen zu können, müssen Sie ein *QueryOptions*-Objekt erstellen und parametrieren:

```
QueryOptions queryOptions = new QueryOptions(CommonFileQuery.DefaultQuery, null);
queryOptions.FolderDepth = FolderDepth.Deep;
queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable;
```

Mit den *QueryOptions* wird eine Query erstellt, die dann mit *GetFilesAsync* abgefragt wird:

```
var query =
    ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions);
foreach (var f in await query.GetFilesAsync())
    ListBox1.Items.Add(f.Path);
```

Dateisuche bedeutet auch fast immer die Verwendung von Filtern. Über das *QueryOption-Objekt* können Sie auch diese Funktionalität realisieren:

```
ListBox1.Items.Add("---------- Rekursiv mit Filter ------------");
queryOptions = new QueryOptions(CommonFileQuery.DefaultQuery,
                                new string[] {".txt"});
```

Sie können hier auch mehrere Dateityp-Filter verwenden, diese dürfen jedoch keine Platzhalter verwenden. Verwenden Sie einfach die Eigenschaft *ApplicationSearchFilter*, um einen Filter mit Platzhalter zu definieren.

```
queryOptions.FolderDepth = FolderDepth.Deep;
queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable;
query =
    ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions);
```

BEISPIEL 21.6: Dateien auflisten

Anzeige:

```
            foreach (var f in await query.GetFilesAsync())
                ListBox1.Items.Add(f.Path);
...
```

Die Vorgehensweise beim Suchen aller Word-Dateien der letzten 14 Tage, mit der Bezeichnung "Kündigung" im Dateinamen und sortiert nach Datum, stellt sich wie folgt dar:

1. Datei-Filter bestimmen:

```
QueryOptions queryOptions = new QueryOptions(CommonFileQuery.DefaultQuery,
                                             new string[] { ".doc", ".docx" });
```

2. Unterverzeichnisse durchsuchen:

```
queryOptions.FolderDepth = FolderDepth.Deep;
```

3. Möglichst Indexer verwenden:

```
queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable;
```

4. Dateinamensfilter anwenden:

```
queryOptions.ApplicationSearchFilter = "*Kündigung*";
```

5. Abfrage erstellen:

```
var query = ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions);
```

6. Dateiliste abrufen:

```
var liste = await query.GetFilesAsync();
```

7. Liste filtern (14 Tage) und sortieren nach Erstellungsdatum:

```
foreach (var f in liste.Where(f => (System.DateTime.Now -
                 f.DateCreated).Days <= 14 ).OrderBy(f => f.DateCreated))
   ListBox1.Items.Add(f.Path);
```

HINWEIS: Die Funktion sucht natürlich nur im App-Datenverzeichnis, eine komplette Laufwerkssuche ist für WinRT-Apps so nicht möglich.

Mögliche Abfragetypen, die Sie mit *CommonFileQuery* erstellen können, zeigt die folgende Tabelle.

Wert	Beschreibung
DefaultQuery	Eine Liste aller Dateien des aktuellen Ordners.
OrderByName	Eine Liste aller Dateien des aktuellen und aller untergeordneter Ordner sortiert nach dem Dateinamen (*ItemNameDisplay*).

Wert	Beschreibung
OrderByTitle	Eine Liste aller Dateien des aktuellen und aller untergeordneter Ordner sortiert nach dem Titel (*Title*). Dies funktioniert jedoch nur in Libraries und im *HomeGroup*-Ordner.
OrderByMusicProperties	Eine Liste aller Dateien des aktuellen und aller untergeordneter Ordner sortiert nach den Musikeigenschaften (nur in Libraries und im *HomeGroup*-Ordner).
OrderBySearchRank	Eine Liste aller Dateien des aktuellen und aller untergeordneter Ordner sortiert nach der Bewertung (*Rank*) und dem Änderungsdatum.
OrderByDate	Eine Liste aller Dateien des aktuellen und aller untergeordneter Order sortiert nach dem Datum.

21.1.5 Dateien erstellen/schreiben/lesen

Der eine oder andere wird vielleicht über die Überschrift gestolpert sein, warum steht da extra "erstellen"? Die Antwort findet sich beim Blick auf die Vorgehensweise beim Schreiben einer Datei:

1. einen *StorageFolder* abrufen
2. ein *StorageFile*-Objekt erstellen oder abrufen
3. Daten in das *StorageFile*-Objekt schreiben

Bei Punkt 2 wird bereits im Zweifelsfall eine leere Datei mit dem angegebenen Namen erzeugt. Später wird mit dem erstellten *StorageFile*-Objekt weitergearbeitet (Öffnen von Lesezugriff oder Schreiblesezugriff).

HINWEIS: Grundsätzlich sollten Sie prüfen, ob Sie überhaupt Daten in einer Datei ablegen. Für Programmeinstellungen etc. ist die Klasse *ApplicationDataContainer* (siehe dazu Abschnitt 20.3.7, Seite 1023) viel besser geeignet. Auch um das Aufräumen bei einer möglichen Deinstallation müssen Sie sich so nicht kümmern.

Datei erstellen

Verwenden Sie dazu die *CreateFileAsync*-Methode des *StorageFolder*-Objekts. Das Verzeichnis ist damit bereits definiert, was bleibt ist zunächst lediglich die Übergabe eines Dateinamens. Optional können Sie angeben was passiert, wenn die Datei bereits existiert. Nutzen Sie dazu die folgenden Werte von *CreationCollisionOption* (siehe folgende Tabelle).

Wert	Beschreibung
GenerateUniqueName	Erstellt in jedem Fall eine neue (leere) Datei. Falls eine gleichnamige Datei vorhanden ist, wird der Name um eine angehängte Ziffer ergänzt, um einen eindeutigen Namen zu ermitteln. Sie sollten bei dieser Variante immer den endgültigen Dateinamen bestimmen, dieser kann ja aus genannten Gründen von Ihrer Vorgabe abweichen.
ReplaceExisting	Erstellt eine neue leere Datei, potenzielle Vorgänger werden gelöscht.
FailIfExists	Beim Versuch eine Datei zu erstellen, die es schon gibt, tritt ein Fehler auf.
OpenIfExists	Es wird eine neue Datei erstellt, wenn diese noch nicht vorhanden ist, andernfalls wird die vorhandene Datei lediglich geöffnet.

HINWEIS: Gerade die letzte Option enthebt Sie der Mühe, mit *FileExists* etc. lange zu prüfen, ob Sie eine Datei öffnen oder neu erstellen müssen.

BEISPIEL 21.7: Eine neue leere Datei erstellen

C#
```
...
using Windows.Storage;
...
       async private void Button_Click_7(object sender, RoutedEventArgs e)
        {
            StorageFolder folder = ApplicationData.Current.LocalFolder;
            StorageFile file = await folder.CreateFileAsync("daten.txt",
                                            CreationCollisionOption.OpenIfExists);
        }
...
```

Schreibzugriff mit FileIO

Sehen wir uns zunächst den einfachsten Dateityp an, den Sie sicher schon an der einen oder anderen Stelle eingesetzt haben: die gute alten Textdatei.

Der schnellste Weg zur fertigen Datei führt über die statische *FileIO*-Klasse. Diese bringt von Haus aus bereits einen ganzen Satz nützlicher Methoden mit, um Daten schnell und ohne Aufwand an eine Datei anzuhängen oder in eine Datei zu schreiben.

BEISPIEL 21.8: Einen Text an eine Textdatei anhängen

C#
```
...
using Windows.Storage;
...
            StorageFolder folder = ApplicationData.Current.LocalFolder;
            StorageFile file = await folder.CreateFileAsync("daten.txt",
                                             CreationCollisionOption.OpenIfExists);
            await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... ");
...
```

HINWEIS: Möchten Sie echte Zeilen auf diese Weise schreiben, müssen Sie den Strings noch jeweils einen Zeilenumbruch hinzufügen (\r\n).

```
            await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
            await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
```

Folgende Methoden stehen alternativ zur Auswahl:

- *AppendLinesAsync*
- *AppendTextAsync*
- *ReadBufferAsync*
- *ReadLinesAsync*
- *ReadTextAsync*
- *WriteBufferAsync*
- *WriteLinesAsync*
- *WriteTextAsync*

Der Sinn der einzelnen Methoden dürfte sich aus der Bezeichnung ergeben.

Schreibzugriff mit StreamWriter

Etwas komplizierter geht es auch mit einem *StreamWriter*, in diesem Fall werden die Daten zunächst überschrieben und dann angehängt.

BEISPIEL 21.9: Schreiben einer Textdatei per *StreamWriter*

C#
```
...
            StorageFolder folder = ApplicationData.Current.LocalFolder;
            StorageFile file = await folder.CreateFileAsync("daten.txt",
                                    CreationCollisionOption.OpenIfExists);
            using (StreamWriter writer = new StreamWriter(await
                                            file.OpenStreamForWriteAsync()))
            {
                await writer.WriteLineAsync("Noch mehr Zeilen (1)");
                await writer.WriteLineAsync("Noch mehr Zeilen (2)");
                await writer.WriteLineAsync("Noch mehr Zeilen (3)");
                await writer.WriteLineAsync("Noch mehr Zeilen (4)");
                await writer.WriteLineAsync("Noch mehr Zeilen (5)");
                await writer.WriteLineAsync("Noch mehr Zeilen (6)");
            }
...
```

Lesezugriff mit FileIO

Auch hier können wir uns mit Hilfe der obigen *FileIO*-Methoden viel Arbeit sparen. Verwenden Sie beispielsweise *ReadLineAsync*, können Sie das Rückgabeergebnis gleich als *IEnumerable* auswerten und zum Beispiel durch die Zeilen iterieren.

BEISPIEL 21.10: Textdatei aus dem Installationsverzeichnis komplett einlesen

C#

```
...
using Windows.Storage;
...
        async private void Button_Click_8(object sender, RoutedEventArgs e)
        {
```

Installationspfad bestimmen:

```
            StorageFolder folder =
                  Windows.ApplicationModel.Package.Current.InstalledLocation;
            StorageFile file = await folder.GetFileAsync("TextFile.txt");
```

Die komplette Datei in eine *IList<string>* einlesen:

```
            var inhalt = await FileIO.ReadLinesAsync(file);
```

Zeilen anzeigen:

```
            foreach (var zeile in inhalt)
                ListBox1.Items.Add(zeile);
...
```

HINWEIS: Alternativ können Sie mit *ReadText* auch gleich die ganze Datei in einen String einlesen.

Lesezugriff mit StreamReader

Auch in der WinRT-Welt findet sich ein *StreamReader,* mit dem zum Beispiel ein zeilenweises Einlesen von Textdateien möglich ist:

BEISPIEL 21.11: Textdatei zeilenweise per *StreamReader* einlesen

C#

```
...
          StorageFolder folder = Windows.ApplicationModel.Package.Current.InstalledLocation;
          StorageFile file = await folder.GetFileAsync("TextFile.txt");
          string zeile = "";
```

StreamReader erzeugen:

```
          using (StreamReader reader = new StreamReader(
                                        await file.OpenStreamForReadAsync()))
          {
```

Zeilenweise einlesen:

```
               while ((zeile = await reader.ReadLineAsync()) != null)
```

BEISPIEL 21.11: Textdatei zeilenweise per *StreamReader* einlesen

```
                    ListBox1.Items.Add(zeile);
            }
...
```

HINWEIS: Auf alle Aspekte des Schreib-Lesezugriffs können wir an dieser Stelle nicht eingehen. Wer beispielsweise einen Schreib-/Lesezugriff auf eine Datei benötigt, sollte sich *IRandomAccessStream* näher ansehen.

21.1.6 Dateien kopieren/umbenennen/löschen

Haben Sie erst einmal ein *StorageFile*-Objekt erstellt, so können Sie mit Hilfe der *CopyAsync*-Methode die Datei relativ problemlos kopieren. "Relativ" deshalb, weil Sie bei der Wahl des Ziels recht vorsichtig sein müssen – Sie erinnern sich, Ihre App läuft nur unter Aufsicht und der Zugriff auf das Dateisystem ist recht eingeschränkt.

BEISPIEL 21.12: Datei kopieren

C#

```
...
        async private void Button_Click_9(object sender, RoutedEventArgs e)
        {
```

Quell-Verzeichnis bestimmen:

```
            StorageFolder folder = ApplicationData.Current.LocalFolder;
```

Quelldatei bestimmen:

```
            StorageFile file = await folder.CreateFileAsync("demodatei.dat");
```

In diesem Fall Zielverzeichnis erzeugen:

```
            StorageFolder zielfolder = await folder.CreateFolderAsync("Backup");
```

Datei kopieren:

```
            await file.CopyAsync(zielfolder,  DateTime.Now.ToString("d") +
                            "_demodatei.dat", NameCollisionOption.ReplaceExisting);
        }
```

Die verschiedenen Überladungen der Methode *CopyAsync* ermöglichen die Übergabe des Zielordners und optional die Angaben, wie die Datei heißen soll und was bei vorhandenen Dateien passieren soll. Verwenden Sie dazu die Werte von *NameCollisionOption*:

Wert	Beschreibung
GenerateUniqueName	Falls Zieldatei bereits vorhanden ist, wird eine neue Datei erstellt, an deren Namen eine laufende Ziffer angehängt wird.
ReplaceExisting	Die Zieldatei wird überschrieben (Rechte vorausgesetzt).
FailIfExists	Die Kopieraktion wird mit einem Fehler abgebrochen.

Ein Sonderfall des Kopierens, das Verschieben, ist per *MoveAsync* bzw. *MoveAndReplaceAsync* möglich. Letzteres ersetzt direkt eine spezifizierte Datei (per *StorageFile*-Parameter), bei *MoveAsync* haben Sie die gleichen Möglichkeiten wie bei einem *CopyAsync*, nur dass diesmal die Quelldatei gelöscht wird. Final löschen können Sie eine Datei mit *DeleteAsync*.

BEISPIEL 21.13: Datei verschieben

C#

```
...
        async private void Button_Click_10(object sender, RoutedEventArgs e)
        {
```

Quellverzeichnis ermitteln:

```
            StorageFolder folder = ApplicationData.Current.LocalFolder;
```

Datei öffnen:

```
            StorageFile file = await folder.CreateFileAsync("demodatei.dat",
                                        CreationCollisionOption.ReplaceExisting);
```

Zielverzeichnis erzeugen oder öffnen:

```
            StorageFolder zielfolder = await folder.CreateFolderAsync("Backup",
                                        CreationCollisionOption.OpenIfExists);
```

Datei verschieben:

```
            await file.MoveAsync(zielfolder, DateTime.Now.ToString("d") + "_demodatei.dat",
                                 NameCollisionOption.ReplaceExisting);
        }
...
```

HINWEIS: Möchten Sie eine Datei umbenennen, so verwenden Sie die Methode *RenameAsync*, auch hier können Sie neben dem obligatorischen Dateinamen optional auch eine *NameCollisionOption* angeben.

21.1.7 Verwenden der Dateipicker

Sicher haben Sie sich schon gefragt, ob die wenigen Verzeichnisse, die wir bisher besprochen haben, die einzigen sind, auf die Sie final zugreifen können. Prinzipiell gilt diese Einschränkung, aber ...

Mit Hilfe eines Nutzereingriffs in Form spezieller Dateidialoge (zu neudeutsch *FilePicker*) können die bisherigen Einschränkungen aufgehoben werden. Dateien, die von diesen Dialogen bzw. dem Nutzer ausgewählt werden, sind jetzt auch für Ihre App sichtbar und entsprechend nutzbar.

Drei FilePicker stehen als Ersatz für die guten alten Open-/Save-Dialoge der Win32-Welt bereit:

- *FileOpenPicker*
- *FileSavePicker*
- *FolderPicker*

Alle drei Picker haben die "unangenehme" Eigenschaft aller WinRT-Apps: sie öffnen sich im Vollbild. Auf einem 10"-Tablet ist dies ja noch verständlich, aber auf einem 27" Display hört der Spaß auf und Sie stehen plötzlich vor einer "Wüste" von Icons und Informationen die mehr verwirrt als nützt.

FileOpenPicker

Zum Öffnen von vorhandenen Dateien nutzen Sie, wie wohl schon vermutet, den *FileOpenPicker*. Fügen Sie dem Picker vor dem Aufruf eine Liste von zulässigen Datei-Extensions (z.B. ".png") hinzu und rufen Sie nachfolgend die Methode *PickSingleFileAsync* oder *PickMultipleFilesAsync* auf.

HINWEIS: Leider ist eine Auswahl mit Platzhalter (z.B. "*icon*.png") nicht möglich, möchten Sie alle Dateien zur Auswahl anbieten, geben Sie als Filter "*" an. Wichtig: Sie müssen wenigstens einen Filter definieren.

BEISPIEL 21.14: Dateiauswahl mit *FileOpenPicker*

C#

```
...
using Windows.Storage.Pickers;
...
        async private void Button_Click_11(object sender, RoutedEventArgs e)
        {
```

Picker instanziieren:

```
            FileOpenPicker fileopen = new FileOpenPicker();
```

Filter festlegen:

```
            fileopen.FileTypeFilter.Add(".png");
            fileopen.FileTypeFilter.Add(".bmp");
```

Dialog öffnen:

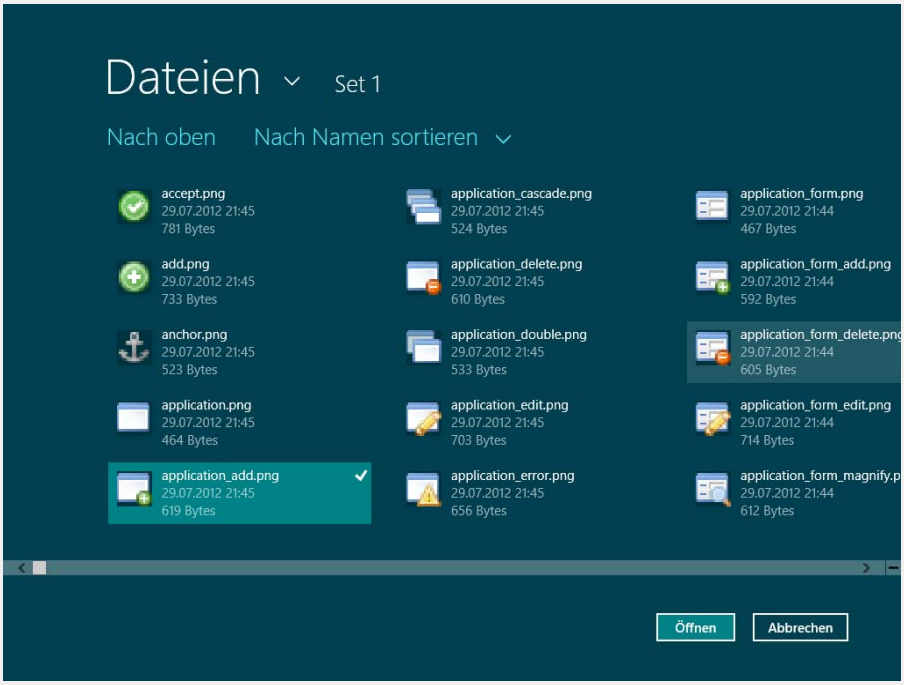
```
            StorageFile file = await fileopen.PickSingleFileAsync();
```

Auswerten:

```
            if (file != null)
            {
                ListBox1.Items.Add("Auswahl: " + file.Path);
                await file.CopyAsync(ApplicationData.Current.LocalFolder);
            }
        }
...
```

BEISPIEL 21.14: Dateiauswahl mit *FileOpenPicker*

Ergebnis



Der *FileOpenPicker* lässt neben der bereits im letzten Beispiel gezeigten Listenansicht auch eine Thumbnail-Ansicht zu. Sie können per Code (*ViewMode*-Eigenschaft) zwischen diesen beiden Ansichten wählen, der Nutzer kann dies später jedoch nicht mehr ändern.

BEISPIEL 21.15: Mehrfachauswahl mit *FileOpenPicker*

C#

```
...
            FileOpenPicker fileopen = new FileOpenPicker();
            fileopen.FileTypeFilter.Clear();
            fileopen.FileTypeFilter.Add(".jpg");
```

Thumbnail-Ansicht:

```
            fileopen.ViewMode = PickerViewMode.Thumbnail;
```

Startverzeichnisauswahl:

```
            fileopen.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
```

Beschriftung der "Öffnen"-Schaltfläche:

```
            fileopen.CommitButtonText = "Diese Datei in die ewigen Jagdgründe befördern";
```

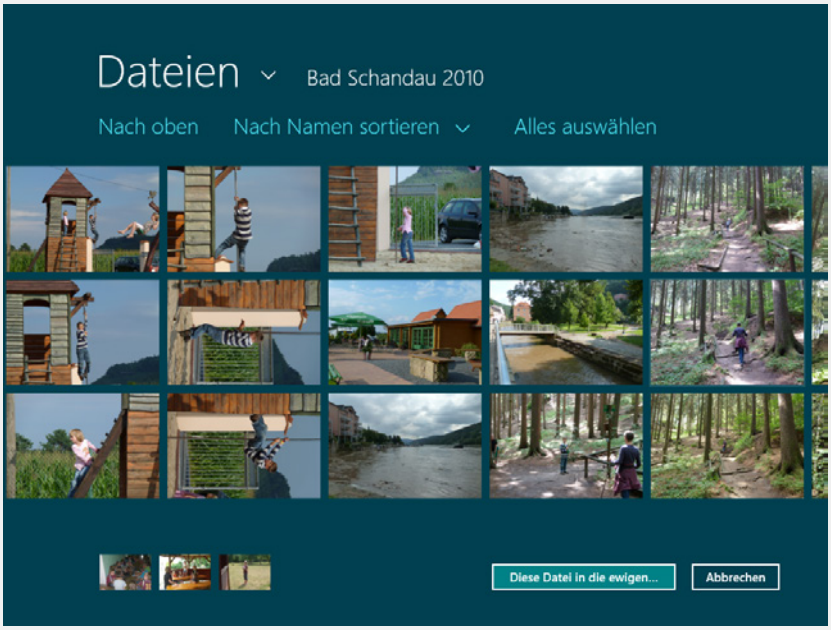
Dateiauswahl:

```
            var files = await fileopen.PickMultipleFilesAsync();
            if (files != null)
```

BEISPIEL 21.15: Mehrfachauswahl mit *FileOpenPicker*

```
                foreach (var file in files)
                    ListBox1.Items.Add("Auswahl: " + file.Path);
...
```

Ergebnis



Wie im obigen Beispiel gezeigt, können Sie mit *SuggestedStartLocation* auch den Startordner für die Auswahl festlegen. Die *PickerLocationId* Enumeration enthält Werte für die wichtigsten Verzeichnisse. Ist Ihnen die Beschriftung der Öffnen-Schaltfläche zu trivial, können Sie über *CommitButtonText* den Text ändern (siehe Beispiel). Last but not least, können Sie auch mehrere Dateien auswählen, rufen Sie dazu *PickMultipleFilesAsync* auf.

HINWEIS: Dateien können nicht nur an einem Speicherort ausgewählt werden, nach der Auswahl einer Datei kann der Nutzer auch in andere Verzeichnisse wechseln und dort Dateien auswählen. Stellen Sie sich also im Zweifel darauf ein, Dateien aus verschiedenen Quellen auswerten zu müssen.

Wird der Dialog mehrfach aufgerufen werden Sie feststellen, dass beim zweiten Start nicht das per *SuggestedStartLocation* angegebene Verzeichnis ausgewählt ist, sondern das beim letzten Aufruf aktive Verzeichnis. Über die *SettingsIdentifier*-Eigenschaft können Sie diese Information zwischen mehreren Dialoginstanzen teilen.

Haben Sie Dateien einmal ausgewählt, erhalten Sie auch vollen Zugriff darauf. Dies gilt prinzipiell so lange wie Sie die entsprechenden Instanzen der *StorageFile*- Objekte besitzen. Wenn diese später einmal benötigt werden hilft die *StorageApplicationPermissions.FutureAccessList* weiter (siehe Seite 1069).

FileSavePicker

Wer eine Datei neu anlegen will, muss sich näher mit dem *FileSavePicker* beschäftigen. Die grundsätzliche Vorgehensweise ist die gleiche wie beim *OpenFilePicker*, nur dass Sie in diesem Fall per *FileTypeChoices* mögliche Dateiextensions vorgeben müssen. Die Standardauswahl für die Dateiextension bestimmen Sie mit *DefaultFileExtension*, einen möglichen Dateinamen können Sie per *SuggestedFileName* vorgeben. Die Anzeige des Dialogs erfolgt mit *PickSaveFileAsync*.

HINWEIS: Die Methode *PickSaveFileAsync* liefert ein *StorageFile*-Objekt zurück, dieses ist zu diesem Zeitpunkt bereits mit einer neuen, leeren Datei verbunden!

BEISPIEL 21.16: Datei mit *FileSavePicker* speichern

C#

```
...
using Windows.Storage.Pickers;
...
        async private void Button_Click_12(object sender, RoutedEventArgs e)
        {
```

Instanz erzeugen:

```
            FileSavePicker filesave = new FileSavePicker();
```

Dateitypauswahl festlegen (diese filtern auch die Ansicht):

```
            filesave.FileTypeChoices.Add("Vektorgrafik",
                                         new List<string>() {".wmf", ".emf" });
            filesave.FileTypeChoices.Add("Pixelgrafik",
                                          new List<string>()  { ".jpg", ".png" });
```

Die Standard-Extension festlegen:

```
            filesave.DefaultFileExtension = ".wmf";
```

Einen Filenamen vorgeben (dieser kann editiert werden):

```
            filesave.SuggestedFileName = "Karte.wmf";
```

Dialog anzeigen:

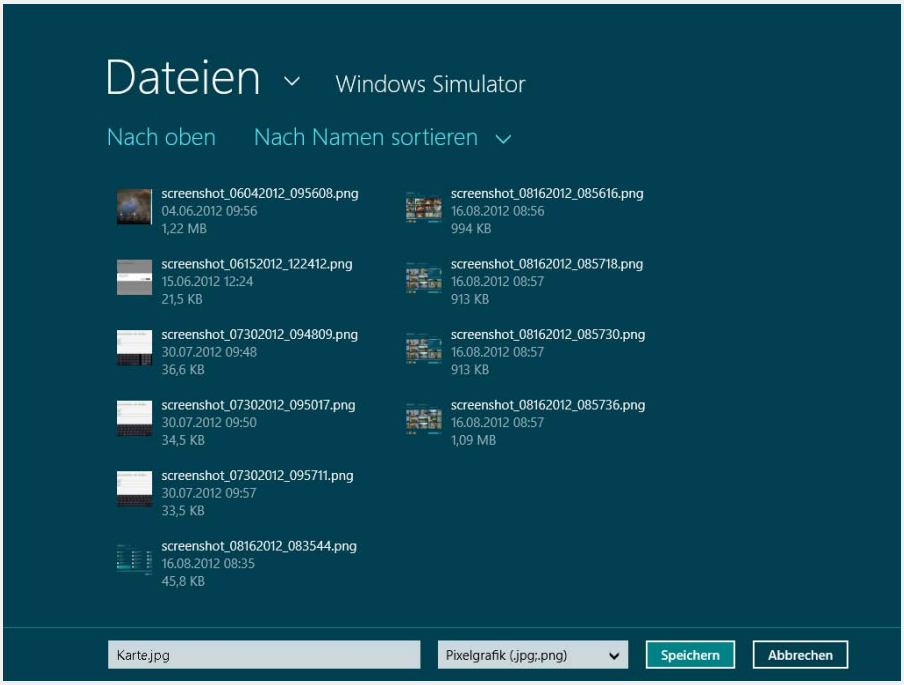
```
            StorageFile file = await filesave.PickSaveFileAsync();
```

Rückgabewert auswerten:

```
            if (file != null)
            {
                ListBox1.Items.Add("Auswahl: " + file.Path);
                await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
            }
        }
...
```

BEISPIEL 21.16: Datei mit *FileSavePicker* speichern

Ergebnis



FolderPicker

WinRT bietet neben der Dateiauswahl auch eine Verzeichnisauswahl an, die Sie einsetzen können, um ein gültiges *StorageFolder*-Objekt eines beliebigen Verzeichnisses zu erhalten. Nachfolgend hat Ihre App vollen Zugriff auf dieses Verzeichnis.

HINWEIS: Voller Zugriff meint auch, dass Ihre App jetzt Dateien löschen kann, die eigentlich nicht im Zugriff Ihrer App stehen.

BEISPIEL 21.17: Einen *FolderPicker* zur Verzeichnisauswahl einsetzen

C#

```
...
      async private void Button_Click_13(object sender, RoutedEventArgs e)
       {
```

Instanz erzeugen:

```
          FolderPicker folderpic = new FolderPicker();
```

Alle Dateien anzeigen:

```
          folderpic.FileTypeFilter.Add("*");
```

BEISPIEL 21.17: Einen *FolderPicker* zur Verzeichnisauswahl einsetzen

Listenansicht auswählen:

```
folderpic.ViewMode = PickerViewMode.List;
```

Wir starten in der Bild-Bibliothek:

```
folderpic.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
```

Dialog anzeigen:

```
StorageFolder folder = await folderpic.PickSingleFolderAsync();
```

Dialog auswerten:

```
if (folder != null)
{
```

Eine Datei im ausgewählten Verzeichnis erstellen:

```
    StorageFile file = await folder.CreateFileAsync("daten.txt",
                                CreationCollisionOption.OpenIfExists);
```

Wir füllen die Datei mit sinnlosem Inhalt:

```
    await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
    await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
    await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... \r\n");
```

Alle Dateien dieses Verzeichnisses anzeigen:

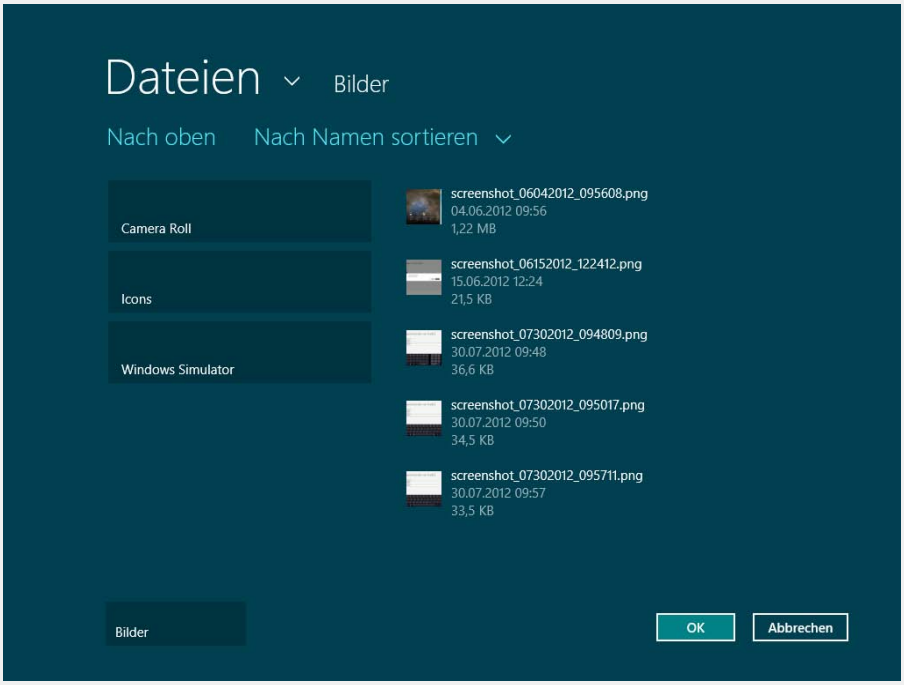
```
    foreach (var f in await folder.GetFilesAsync())
    {
        ListBox1.Items.Add(f.Path);
```

Auch das geht jetzt – eine Datei löschen:

```
        if (f.Name == "Test.txt")
            await f.DeleteAsync();
    }
}
}
...
```

BEISPIEL 21.17: Einen *FolderPicker* zur Verzeichnisauswahl einsetzen

Ergebnis



21.1.8 StorageFile-/StorageFolder-Objekte speichern

Haben Sie sich schon mal gefragt was passiert, wenn Sie zum Beispiel nochmal auf ein Verzeichnis oder eine Datei zugreifen wollen, die schon einmal per Picker ausgewählt wurde? Im Normalfall haben Sie darauf keinen Zugriff, doch mit Hilfe der *FutureAccessList* (Klasse *StorageApplicationPermissions*) lässt sich diese Einschränkung umgehen. Sie speichern quasi die aktuellen Rechte in dieser Liste ab und können später wieder darauf zugreifen.

BEISPIEL 21.18: Verwendung der *FutureAccessList*

C#

```
...
using Windows.Storage.AccessCache;
...
```

Stellen Sie sich vor, Sie rufen mit *PickSingleFolderAsync* einen Zielordner auf (z.B. den Desktop):

```
StorageFolder folder = await folderpic.PickSingleFolderAsync();
if (folder != null)
{
...
```

BEISPIEL 21.18: Verwendung der *FutureAccessList*

An dieser Stelle speichern Sie den *StorageFolder* in der *FutureAccessList* ab:

```
string token = StorageApplicationPermissions.FutureAccessList.Add(
                                folder, folder.Path);
```

Übergeben Sie das Objekt und optional einen frei wählbaren Bezeichner (wir nehmen den Pfad). Rückgabewert ist eine eindeutige ID (GUID) für unseren Eintrag.

```
ListBox1.Items.Add("Token: " + token);
...
```

Zu einem späteren Zeitpunkt (auch nach einem Neustart) können Sie die *FutureAccessList* abrufen und der App den Zugriff auf die gespeicherten Objekte ermöglichen:

```
async private void Button_Click_14(object sender, RoutedEventArgs e)
{
    ListBox1.Items.Add("Vorhandene FutureAccessList-Entries");
    foreach (var item in StorageApplicationPermissions.FutureAccessList.Entries)
    {
```

Sie können in der Liste die Metadaten (freier Bezeichner) und die eindeutige ID abrufen:

```
ListBox1.Items.Add("Metadata: " + item.Metadata + " Token: " + item.Token);
```

Hier rufen wir per ID das *StorageFolder*-Objekt wieder aus der Liste ab:

```
StorageFolder folder =  await
  StorageApplicationPermissions.FutureAccessList.GetFolderAsync(item.Token);
```

Nachfolgend können Sie mit dem Ordner so arbeiten, als ob Sie gerade den *FolderPicker* verwendet haben (Vollzugriff). Im Beispiel erzeugen wir eine Datei:

```
StorageFile file = await folder.CreateFileAsync("daten.txt",
                                    CreationCollisionOption.OpenIfExists);
await FileIO.WriteTextAsync(file, "Ich wurde gerade verändert!");
    }
}
```

Haben Sie die IDs abgespeichert, können Sie über diese auf einen Eintrag testen und im Erfolgsfall den Eintrag wieder herstellen:

```
if (StorageApplicationPermissions.FutureAccessList.ContainsItem(<GespeicherteID>))
{
    StorageFolder folder =  await
           StorageApplicationPermissions.FutureAccessList.GetFolderAsync(<GespeicherteID>);
}
```

HINWEIS: Verwenden Sie die Einträge aus der Liste, kommen Sie ganz ohne Nutzereingriff aus.

Einen einmal erteilten Zugriff auf die Objekte können Sie durch Löschen der Einträge (*Remove* oder *Clear*) widerrufen.

HINWEIS: Sinngemäß können Sie auf diese Weise natürlich auch den Zugriff auf einzelne Dateien (*StorageFile*-Objekte) sichern. Rufen Sie in diesem Fall das Objekt mit *GetFileAsync* ab.

Interessant ist noch die Methode *AddOrReplace*, hier wird ein eventuell vorhandener Eintrag nur aktualisiert, andernfalls wird ein neuer Eintrag erzeugt. Voraussetzung für diese Vorgehensweise ist jedoch, dass Sie die eindeutige ID gespeichert haben, da diese für den Vergleich genutzt wird.

Achtung:

HINWEIS: Sie können nicht unbegrenzt viele Einträge in der Liste speichern. Die Eigenschaft *MaximumItemsAllowed* gibt Ihnen die maximal mögliche Zahl von Einträgen zurück, gehen Sie also sparsam damit um[1].

21.1.9 Verwenden der Most Recently Used-Liste

Benötigen Sie in Ihren Programmen eine Liste der zuletzt geöffneten Dateien? Vielleicht sind Sie schon auf die Idee gekommen, die *FutureAccessList* dafür zu missbrauchen. Sparen Sie sich dahingehende Überlegungen, genau für diesen Einsatzzweck steht Ihnen eine weitere, fast funktionsgleiche Liste zur Verfügung: *StorageApplicationPermissions.MostRecentlyUsedList*. Fügen Sie dieser Liste einfach per *Add* neue Einträge hinzu, diese werden automatisch an den Anfang der Liste gestellt. Überschreitet die Liste das Maximum (derzeit 25 Einträge), werden ältere Einträge gelöscht.

HINWEIS: Einziger Unterschied zur *FutureAccessList* ist ein zusätzliches Ereignis *ItemRemoved*, das jedoch nicht, wie vielleicht vermutet, beim Löschen der älteren Einträge aufgerufen wird.

BEISPIEL 21.19: Eine Liste von Dateien zur MRU hinzufügen

C#
```
...
using Windows.Storage;
using Windows.Storage.Pickers;
using Windows.Storage.AccessCache;
...
        async private void Button_Click_15(object sender, RoutedEventArgs e)
        {
            FileOpenPicker fileopen = new FileOpenPicker();
            fileopen.FileTypeFilter.Add("*");
            fileopen.ViewMode = PickerViewMode.Thumbnail;
            fileopen.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
            fileopen.CommitButtonText = "Zur MRU hinzufügen";
```

[1] Sie können ja den Anwender "überreden" die Root von Laufwerk "C:" auszuwählen ...

BEISPIEL 21.19: Eine Liste von Dateien zur MRU hinzufügen

Wir wählen per Picker mehrere Dateien aus:

```
            var files = await fileopen.PickMultipleFilesAsync();
            if (files != null)
            {
```

Einfügen in die MRU-Liste (wir fügen einen Zeitstempel mit ein, so sehen Sie gleich, wie die Einträge abgelegt werden):

```
                foreach (var file in files)
                {
                    StorageApplicationPermissions.MostRecentlyUsedList.Add(file,
                        DateTime.Now.ToString() + "  " + file.Path);
                }
            }
        }
```

Das spätere Auslesen:

```
        private void Button_Click_16(object sender, RoutedEventArgs e)
        {
            ListBox1.Items.Clear();
            foreach (AccessListEntry entry in
                        StorageApplicationPermissions.MostRecentlyUsedList.Entries)
            {
                ListBox1.Items.Add(entry.Metadata);
            }
        }
...
```

Damit schließen wir die Thematik "Dateien und Verzeichnisse" ab und wenden uns einigen neuen Konzepten der WinRT-Programmierung zu.

21.2 Datenaustausch zwischen Apps/Programmen

Ein Programm bzw. eine App steht meist nicht allein und so sind Sie in vielen Fällen darauf angewiesen, Daten zwischen verschiedenartigen Applikationen auszutauschen. Sei es ein Bild, eine Datei, eine Information aus einer Datenbank oder ähnliches. Gerade WinRT stellt in diesem Bereich einige für den Desktop-Programmierer gänzliche neue Konzepte bereit, die aus Anwendersicht die strikte Trennung zwischen verschiedenen Apps weitgehend aufheben.

Doch bevor wir diese besprechen, wollen wir uns mit einem alten Bekannten vertraut machen, der uns schon seit den Anfangstagen der Windows-Programmierung begleitet.

21.2.1 Zwischenablage

Auch wenn es für WinRT-Apps bereits effizientere Weg zum Datenaustausch gibt – ein Hauptvorteil der guten alten Zwischenablage bleibt, dass Sie über diese auch Daten mit konventionellen Desktop-Programmen austauschen können.

HINWEIS: Ihre App sollte, wo immer es intuitiv und sinnvoll erscheint, Zwischenablagefunktionen bereitstellen, da dies der bekannteste und universellste Weg des Datenaustauschs unter Windows ist.

Über den Namespace *Windows.ApplicationModel.DataTransfer* steht Ihnen eine statische Klasse *Clipboard* zur Verfügung. Nutzen Sie diese zum ersten Mal, werden Sie sicher Methoden wie *SetText* oder *ContainsText* vermissen. Diese Klasse ist universeller aufgebaut, alle Aktionen laufen über die Methoden *SetContent* und *GetContent*. Während erstere ein *DataPackage* als Parameter erwartet, liefert letztere ein *DataPackageView* als Rückgabewert.

Die Methoden der *DataPackage*-Klasse suggerieren, welche Daten problemlos kopiert werden können:

- *SetBitmap*
- *SetData*
- *SetHtmlFormat*
- *SetRtf*
- *SetStorageItems*
- *SetText*
- *SetUri*

Beim Einfügen von Daten sollten Sie über das *DataPackageView*-Objekt zunächst die verfügbaren Formate bestimmen (Methode *Contains*), bevor Sie auf eine der folgenden Methoden zurückgreifen:

- *GetBitmapAsync*
- *GetDataAsync*
- *GetHtmlFormatAsync*
- *GetResourceMapAsync*
- *GetRtfAsync*
- *GetStorageItemsAsync*
- *GetTextAsync*
- *GetUriAsync*

Einige Beispiele sollen Ihnen die Vorgehensweise demonstrieren.

Texte kopieren und einfügen

Beim Kopieren von Texten brauchen Sie sich nicht auf das reine Textformat beschränken, wo es sinnvoll ist können Sie auch gleich noch die Daten im HTML oder RTF-Format in die Zwischenablage kopieren. Die Zielanwendung entscheidet dann (bzw. der Nutzer) welches der unterstützten Formate eingefügt werden soll.

BEISPIEL 21.20: Kopieren und Einfügen von Texten

C#

```
...
using Windows.ApplicationModel.DataTransfer;
...
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
```

Zunächst unser Text:

```
            string meintext = "Dieser Text stammt aus der App!!!!";
```

Gleiches nochmal im HTML-Format:

```
            string meinhtmltext = "<h1>" + meintext + "</h1><br/> Noch eine Zeile!";
```

Wir erstellen ein *DataPackage*, dieses nimmt die Daten auf:

```
            DataPackage datapackage = new DataPackage();
```

Die Texte zum *DataPackage* hinzufügen:

```
            datapackage.SetText(meintext);
            datapackage.SetHtmlFormat(HtmlFormatHelper.CreateHtmlFormat(meinhtmltext));
```

Mit *HtmlFormatHelper.CreateHtmlFormat* wird das von uns übergebene HTML-Fragment in ein gültiges HTML-Format überführt.

Das eigentliche Kopieren der Daten:

```
            try
            {
                Clipboard.SetContent(datapackage);
                ListBox1.Items.Add("Daten kopiert ...");
            }
            catch (Exception ex)
            {
                ListBox1.Items.Add("Fehler: " + ex.Message);
            }
        }
...
```

Beim Abrufen von Daten lesen wir zunächst ein *DataPackageView*-Objekt ein:

```
        async private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            DataPackageView dpv = Clipboard.GetContent();
```

BEISPIEL 21.20: Kopieren und Einfügen von Texten

Nachfolgend prüfen wir, ob das gewünschte Datenformat (in unserem Fall Text) vorhanden ist:

```
            if (dpv.Contains(StandardDataFormats.Text))
            {
```

Ist dies der Fall, können wir die Daten per *GetTextAsync* abrufen:

```
                string txt = await dpv.GetTextAsync();
                ListBox1.Items.Add(txt);
            }
        }
...
```

HINWEIS: Es liegt nahe, dass Sie den Inhalt der Zwischenablage mit *Clear* löschen.

Bilder kopieren und einfügen

Prinzipiell handelt es sich um die gleiche Vorgehensweise wie bei den Texten, wir müssen uns aber etwas intensiver mit der Aufbereitung der Bilddaten beschäftigen, da die Methode *SetBitmap* die Daten als *RandomAccessStreamReference* erwartet.

BEISPIEL 21.21: Kopieren und Einfügen von Bildern

C#

```
...
using Windows.ApplicationModel.DataTransfer;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Media.Imaging;
...
        private void Button_Click_4(object sender, RoutedEventArgs e)
        {
```

DataPackage erzeugen:

```
            DataPackage datapackage = new DataPackage();
```

Wir laden Daten aus den Programm-Ressourcen, eigentlich handelt es sich um einen Zugriff auf das Installationsverzeichnis:

```
            datapackage.SetBitmap(RandomAccessStreamReference.CreateFromUri(
                                  new Uri("ms-appx:///bulbon.png")));
```

Daten kopieren:

```
            Clipboard.SetContent(datapackage);
        }
...
```

Beim Einfügen der Daten, in diesem Fall in ein *Image*-Control zur Anzeige, müssen wir ebenfalls den Umweg über eine *IRandomAccessStreamReference* nehmen:

BEISPIEL 21.21: Kopieren und Einfügen von Bildern

```
        async private void Button_Click_5(object sender, RoutedEventArgs e)
        {
            DataPackageView dpv = Clipboard.GetContent();
```

Wenn eine Bitmap enthalten ist:

```
            if (dpv.Contains(StandardDataFormats.Bitmap))
            {
                try
                {
```

Bitmap-Inhalte abrufen:

```
                    IRandomAccessStreamReference imgdata = await dpv.GetBitmapAsync();
                    if (imgdata != null)
                    {
```

Ein internes *BitmapImage*-Objekt wird mit den Daten initialisiert und nachfolgend als Quelle für das *Image*-Control genutzt:

```
                        BitmapImage img = new BitmapImage();
                        img.SetSource(await imgdata.OpenReadAsync());
                        Image1.Source = img;
                    }
                }
                catch (Exception ex)
                {
                    ListBox1.Items.Add(ex.Message);
                }
            }
        }
...
```

Dateien kopieren/verschieben und abrufen

Möchten Sie Dateien kopieren oder verschieben, können Sie dies bekanntermaßen auch per Zwischenablage tun. Prinzipiell werden in diesem Fall jedoch nicht die Inhalte der Dateien kopiert, sondern es werden lediglich die Dateinamen und die auszuführende Operation in der Zwischenablage hinterlegt. Das Zielprogramm übernimmt die Dateireferenzen und führt die Operationen entsprechend aus.

BEISPIEL 21.22: Dateien in die Zwischenablage kopieren

C#

```
...
using Windows.ApplicationModel.DataTransfer;
using Windows.Storage.Streams;
using Windows.Storage;
using Windows.Storage.Pickers;
...
```

BEISPIEL 21.22: Dateien in die Zwischenablage kopieren

```
async private void Button_Click_6(object sender, RoutedEventArgs e)
{
```

Zunächst wählen wir einige Dateien aus der Bild-Bibliothek aus:

```
    FileOpenPicker fileopen = new FileOpenPicker();
    fileopen.FileTypeFilter.Add("*");
    fileopen.ViewMode = PickerViewMode.Thumbnail;
    fileopen.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
    fileopen.CommitButtonText = "Zur Zwischenablage hinzufügen";
    var files = await fileopen.PickMultipleFilesAsync();
    if (files != null)
    {
```

Wurden Dateien gewählt, erzeugen wir das bereits bekannte *DataPackage* und übergeben die *StorageFiles*:

```
        var datapackage = new DataPackage();
        datapackage.SetStorageItems(files);
```

Nachfolgend müssen Sie noch die Operation für die Daten bestimmen:

```
        datapackage.RequestedOperation = DataPackageOperation.Copy;
```

Zur Auswahl stehen hier *Copy*, *Move*, *Link* und *None*.

```
        try
        {
```

Daten kopieren:

```
            Clipboard.SetContent(datapackage);
        }
        catch (Exception ex)
        {
            ListBox1.Items.Add(ex.Message);
        }
    }
}
```

Wollen wir selbst Dateien abrufen, prüfen wir zunächst auf das Vorhandensein des richtigen Datenformats:

```
async private void Button_Click_7(object sender, RoutedEventArgs e)
{
    var dpv = Clipboard.GetContent();
    if (dpv.Contains(StandardDataFormats.StorageItems))
    {
        try
        {
```

BEISPIEL 21.22: Dateien in die Zwischenablage kopieren

Wir rufen die Liste der Dateien (oder auch Verzeichnisse) ab:

```
                var files = await dpv.GetStorageItemsAsync();
                if (files != null)
                {
```

Wir bestimmen die auszuführende Operation (hier nur zur Anzeige):

```
                    string todo = dpv.RequestedOperation.ToString();
```

Wir listen die Dateien auf (hier würde eigentlich eine Auswertung erfolgen):

```
                    foreach (var file in files)
                        ListBox1.Items.Add(todo + ": " + file.Path);
                }
            }
            catch (Exception ex)
            {
                ListBox1.Items.Add(ex.Message);
            }
        }
    }
```

HINWEIS: Vergessen Sie nicht, dass neben Dateien auch Verzeichnisse in die Zwischenablage kopiert werden können (die Liste enthält *StorageItem*-Objekte). Eine Unterscheidung zwischen enthaltenen *StorageFile*- und *StorageFolder*-Objekten können Sie mit der *IsOfType*-Methode treffen.

```
...
                var files = await dpv.GetStorageItemsAsync();
                if (files != null)
                {
                    foreach (var file in files)
                    {
                        if (file.IsOfType(StorageItemTypes.Folder))
                            ListBox1.Items.Add("Folder: " + file.Path);
                        ListBox1.Items.Add(todo + ": " + file.Path);
                    }
                }
...
```

Zwischenablage überwachen

Vielleicht kennen Sie diese Funktionalität bereits aus "konventionellen" Programmen, bei denen man sich mühsam in die Message-Verwaltung von Windows einklinken musste, um Änderungen in der Zwischenablage zu überwachen. Nach dem Programmende musste man sich aus der Ereigniskette wieder ausklinken.

All diesen Aufwand brauchen Sie in WinRT-Apps nicht zu treiben, es genügt, wenn Sie das entsprechende Ereignis *ContentChanged* des *Clipboard*-Objekts unterstützen. Dort können Sie die enthaltenen Formate auswerten und Inhalte gegebenenfalls sofort im Programm anzeigen.

HINWEIS: Die Reaktion per Ereignis auf Änderungen der Zwischenablageinhalte verliert in WinRT-Apps an Bedeutung, da Sie nur für die sichtbare Vordergrund-App sicherstellen können, dass auch das Ereignis ausgelöst wird (denken Sie an das Suspending). Kopiert eine andere App/Anwendung Daten in die Zwischenablage, ist diese jedoch meist die Vordergrundanwendung (Ausnahme: Multimonitor-Betrieb).

BEISPIEL 21.23: Reaktion auf Änderungen in der Zwischenablage

C#

```
...
using Windows.ApplicationModel.DataTransfer;
...
   public sealed partial class BasicPage1 : Datenaustausch.Common.LayoutAwarePage
    {
        public BasicPage1()
        {
            this.InitializeComponent();
```

Die Ereignisprozedur zuweisen:

```
            Clipboard.ContentChanged += Clipboard_ContentChanged;
        }
```

Die eigentliche Ereignisprozedur:

```
        void Clipboard_ContentChanged(object sender, object e)
        {
            ListBox1.Items.Add("Änderung in der Zwischenablage: ");
```

Enthaltene Formate auslesen und anzeigen:

```
            DataPackageView dpv = Clipboard.GetContent();
            foreach (var format in dpv.AvailableFormats)
            {
                ListBox1.Items.Add("   verfügbar: " + format.ToString());
            }
        }
...
```

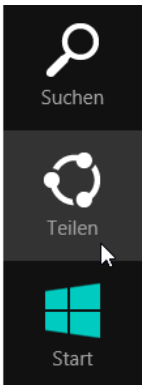
HINWEIS: Hier tut sich dann auch eine kleine Schwachstelle des WinRT-Sicherheitskonzepts auf: Sie überwachen beispielsweise die Zwischenablage und sammeln vielleicht interessante *StorageFile*- oder *StorageFolder*-Objekte ein und speichern diese in der *FutureAccessList*. Später haben Sie dann vollen Zugriff auf den entsprechenden Speicherort (so er noch vorhanden ist).

Kommen wir nun jedoch zu den modernen Nachfolgern der Zwischenablage.

21.2.2 Teilen von Inhalten

Neben der Zwischenablage bietet WinRT mit dem Teilen von Inhalten per *DataTransferManager* eine weitere Möglichkeit, Daten anderen Apps zur Verfügung zu stellen. Haben Sie den vorhergehenden Abschnitt intensiv studiert, werden Sie im Weiteren viele Gemeinsamkeiten vorfinden, wird doch die gleiche *DataPackage*-Klasse für den Datentransport genutzt.

Der prinzipielle Ablauf unterscheidet sich allerdings etwas vom Konzept der Zwischenablage: Das Teilen von Inhalten wird vom Nutzer nicht wie gewohnt über die App initiiert, sondern per Teilen-Funktion auf dem CharmBar:



Ihre App muss/kann auf diese Anfrage des Systems reagieren und Inhalte bereitstellen.

HINWEIS: Leider liegt in diesem Ansatz auch schon das Problem des Konzepts. Haben Sie eine App-Seite mit verschiedenen Informationen bleibt keine Auswahl, welche Daten denn zu kopieren sind. Eventuell werten Sie hier das aktuelle Steuerelement aus.

Was sich etwas kompliziert anhört, reduziert sich auf die Unterstützung des *DataTransferManager.-DataRequested*-Ereignisses. Nachfolgend arbeiten Sie quasi wie mit der Zwischenablage.

BEISPIEL 21.24: Daten mit dem *DataTransferManager* teilen

C#

```
...
using Windows.ApplicationModel.DataTransfer;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Media.Imaging;
using Windows.Storage;
...
    public sealed partial class BasicPage1 : Datenaustausch.Common.LayoutAwarePage
    {
```

Eine lokale Instanz des *DataTransferManager* erzeugen:

```
        DataTransferManager dtm;
```

BEISPIEL 21.24: Daten mit dem *DataTransferManager* teilen

Wird die Seite geöffnet, rufen wir für die aktuelle Ansicht den *DataTransferManager* ab und belegen das *DataRequested*-Ereignis:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    dtm = DataTransferManager.GetForCurrentView();
    dtm.DataRequested += DataRequested;
}
```

Beim Schließen der Seite sollten wir auch das Ereignis wieder "abklemmen":

```
protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    dtm.DataRequested -= new TypedEventHandler<DataTransferManager,
                            DataRequestedEventArgs>(DataRequested);
}
```

Das folgende Ereignis wird durch die Teilen-Funktion auf dem Charmbar aktiviert:

```
async void DataRequested(DataTransferManager sender, DataRequestedEventArgs args)
{
```

Um das *DataPackage* müssen wir uns nicht selbst kümmern, dieses wird gleich per Parameter übergeben:

```
    DataPackage datapackage = args.Request.Data;
```

Wir setzen einen kurzen Titel (wird im Flyout angezeigt):

```
    datapackage.Properties.Title = "Beispiel-APP";
```

Optional können Sie auch eine längere Beschreibung einblenden:

```
    datapackage.Properties.Description = "Daten aus der Beispiel-App";
```

Teilen Sie Bilddaten, können Sie auch eine Thumbnail-Grafik zuweisen, diese kann die Ziel-App anzeigen, bis das eigentliche Bild zur Verfügung steht:

```
    datapackage.Properties.Thumbnail = RandomAccessStreamReference.CreateFromUri(
                        new Uri("ms-appx:///bulbon.png"));
```

Einen Text teilen:

```
    datapackage.SetText("Ein Text aus dem Beispielprogramm, " +
                       "der in andere Apps eingefügt werden kann.");
```

Eine Bitmap kopieren wir gleich mit:

```
    datapackage.SetBitmap(RandomAccessStreamReference.CreateFromUri(
                        new Uri("ms-appx:///bulboff.png")));
```

Alternativ können Sie auch StorageFiles teilen:

```
    List<IStorageItem> items = new List<IStorageItem>();
    items.Add(await Package.Current.InstalledLocation.GetFileAsync("bulbon.png"));
```

BEISPIEL 21.24: Daten mit dem *DataTransferManager* teilen

```
            datapackage.SetStorageItems(items);
        }
...
```

Ergebnis

Nach dem Aufruf der App öffnen Sie den Charmbar und klicken auf die Funktion *Teilen*. Nachfolgend sollte folgendes Flyout zu sehen sein:

Wie Sie sehen, erscheinen die eingestellten Properties Ihrer App so wie eine Liste der Ziel-Apps, die etwas mit den angebotenen Daten anfangen können[1].

Unser Beispiel teilt eigentlich neben dem Text auch eine Datei so wie eine Grafik. Was die Ziel-App mit dieser "Informationsflut" anfängt ist jedoch Ihre Sache. Wählen wir zum Beispiel die Mail-App, wird diese lediglich die Bitmap einfügen, der Text und die Datei werden ignoriert.

Funktion "Teilen" programmgesteuert aufrufen

Nicht in jedem Fall ist es sinnvoll, auf die Kenntnisse des Nutzers zu vertrauen (Charmbar aufrufen, Funktion Teilen anklicken ...). Sie können die Teilen-Funktion auch direkt aus Ihrer App heraus aktivieren, rufen Sie dazu einfach die Methode *ShowShareUI* auf:

BEISPIEL 21.25: Teilen per App aktivieren

C#

```
...
        private void Button_Click_8(object sender, RoutedEventArgs e)
        {
            DataTransferManager.ShowShareUI();
        }
...
```

[1] Im Augenblick ist diese Liste noch sehr überschaubar, im Laufe der Zeit werden sich sicher diverse Apps auf den PCs tummeln.

Sie möchten temporär keine Daten teilen

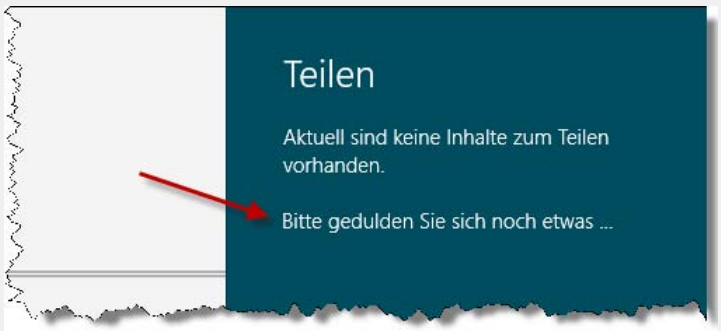
Auch wenn Ihre App das Ereignis *DataRequested* prinzipiell unterstützt kann es möglich sein, dass Ihre App noch nicht zum Datentransfer bereit ist (Daten noch nicht geladen, anderweitig beschäftigt etc.). In diesem Fall sollten Sie eine entsprechende Meldung anzeigen:

BEISPIEL 21.26: Meldung, wenn keine Daten vorhanden sind

C#

```
...
        async void DataRequested(DataTransferManager sender, DataRequestedEventArgs args)
        {
            DataPackage datapackage = args.Request.Data;
            datapackage.Properties.Title = "Beispiel-APP";
            datapackage.Properties.Description = "Daten aus der Beispiel-App";
...
            args.Request.FailWithDisplayText("Bitte gedulden Sie sich noch etwas ...");
        }
...
```

Ergebnis



Damit beenden wir unsere Ausführungen zum Teilen von Daten und sehen uns die Gegenseite, d h. den Empfänger der Daten, näher an.

21.2.3 Eine App als Freigabeziel verwenden

Neben dem Teilen von Daten ist für einige Anwendungstypen auch das Konsumieren geteilter Daten interessant. Prominentestes Beispiel ist die Mail-App, die übergebene Daten (Texte, Bilder oder Dateien) in neue Emails integriert. Sie müssen nur noch die Adressdaten ausfüllen und können schon die Email absenden. Folgende Schritte sind für eine erfolgreiche Integration nötig:

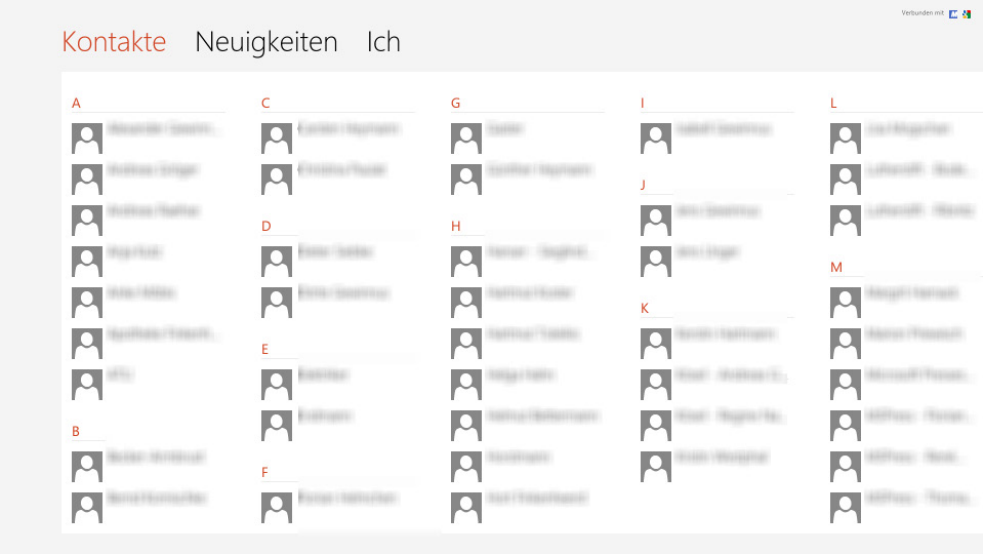
- Registrieren der App als Freigabeziel
- Definieren der unterstützten Daten-/Dateiformate
- Implementieren des Eintrittspunktes *OnShareTargetActivated*
- Definieren einer Seite für die Anzeige und Übernahme der Daten
- Auswerten der Daten

Das sieht schon etwas komplexer aus als bei der Zwischenablage, aber hier handelt es sich ja auch um einen etwas weiter gefassten Ansatz. Aus diesem Grund finden Sie eine komplette Anwendung in den Praxisbeispielen des Kapitels:

► 21.7.2 Eine App als Freigabeziel verwenden

21.2.4 Zugriff auf die Kontaktliste

Windows 8 bietet eine zentrale Kontaktliste, die die Daten verschiedener Konten (Hotmail, GMail etc.) zusammenführen kann. Was liegt näher, als sich in diesem reichhaltigen Fundus zu bedienen, auf diese Weise ersparen Sie dem Nutzer eventuell die mühsame Recherche bei eigenen Eingaben (Telefonnummern, EMail-Adressen) in Ihrer App.



Die Umsetzung entspricht der bei einem *FilePicker*, nur dass wir es jetzt eben mit einem *ContactPicker* zu tun haben.

BEISPIEL 21.27: Verwendung *ContactPicker*

C#

...

Namespace für den *ContactPicker* einbinden:

```
using Windows.ApplicationModel.Contacts;
...
        async private void Button_Click_9(object sender, RoutedEventArgs e)
        {
```

Picker instanziieren:

```
            ContactPicker picker = new ContactPicker();
```

BEISPIEL 21.27: Verwendung *ContactPicker*

Wir möchten mehrere Kontakte abrufen, alternativ können Sie auch die Methode *PickSingleContactAsyc* aufrufen:

```
var contacts= await picker.PickMultipleContactsAsync();
```

Wir werten die Liste der zurückgegebenen Kontakte aus:

```
foreach (var contact in contacts)
{
```

Die Liste besteht zunächst aus der Grundinformation *Name* und zusätzlich aus weiteren Unterlisten (*Emails*, *PhoneNumbers*, *Locations*, *InstantMessages* und *CustomFields*), die Sie ebenfalls auflisten müssen:

```
    ListBox1.Items.Add(contact.Name);
```

Wir belassen es bei der Auswertung von Email-Adressen und Telefonnummern:

```
    foreach (var email in contact.Emails)
        ListBox1.Items.Add("    Email: " + email.Value);
    foreach (var phonenumber in contact.PhoneNumbers)
        ListBox1.Items.Add("    Telefon: " + phonenumber.Category + " " +
                           phonenumber.Value);
```

Das *Value*-Feld enthält die eigentliche Emailadresse bzw. Telefonnummer, die Beschriftung (Privat, Geschäftlich Mobil etc.) findet sich im Feld *Category*.

```
    }
}
...
```

Ergebnis

HINWEIS: Ihre App kann ebenfalls Anbieter von Kontaktinformationen werden, dazu muss der Contract *Inhaltsauswahl* unterstützt werden (*OnActivated*-Methode überschreiben). An dieser Stelle können wir aus Platzgründen auf dieses Thema leider nicht weiter eingehen.

21.3 Spezielle Oberflächenelemente

In den vorhergehenden Kapiteln hatten wir ja bereits die wichtigsten Oberflächen-Elemente von Apps besprochen, an dieser Stelle wollen wir uns nun mit einigen etwas spezielleren Vertretern dieser Gattung beschäftigen.

21.3.1 MessageDialog

Vermutlich ist es Ihnen ähnlich wie den Autoren ergangen: Sie haben die gute alte *MessageBox* schmerzlich vermisst, um zum Beispiel einfach nur einen Wert oder eine Reaktion auf Eingaben anzuzeigen.

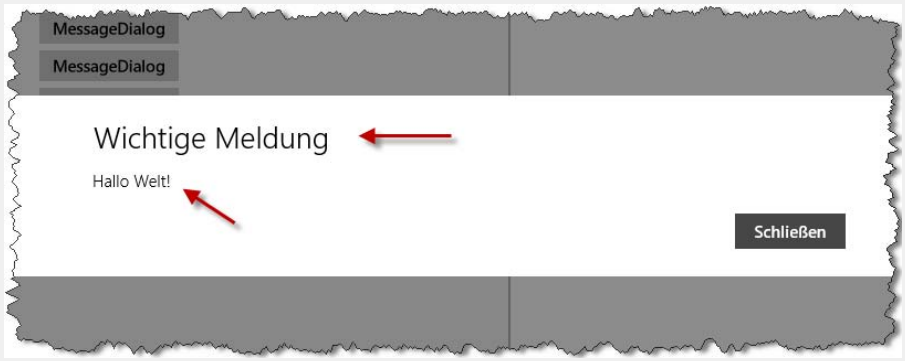
Prinzipiell gibt es die Klasse bzw. deren Funktionalität nicht mehr, Sie müssen sich wohl oder übel mit der neuen Klasse *MessageDialog* anfreunden, die eine ähnliche Funktionalität aufweist.

BEISPIEL 21.28: Einfache Meldung per MessageDialog anzeigen

C#

```
...
using Windows.UI.Popups;
...
        async private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            MessageDialog msgdlg = new MessageDialog("Hallo Welt!", "Wichtige Meldung");
            var res = await msgdlg.ShowAsync();
        }
...
```

Ergebnis



Im Konstruktor übergeben Sie bereits den Meldungstext und die Überschrift des neuen Dialogs, der sich wie ein modales Popup-Fenster verhält. Alternativ können Sie beide Werte auch über die Eigenschaften *Title* und *Content* setzen. Der Aufruf erfolgt mit *ShowAsync*, auch wenn die App-Oberfläche darunter sowieso nicht verwendbar ist.

Dialog mit Auswahl

Meist bleibt es nicht bei einer einfachen Meldung, vielfach sollen auch verschiedene Optionen zur Auswahl stehen. Damit ergeben sich dann auch schon eine Reihe von Fragen:

- Wie erstelle ich weitere Schaltflächen?
- Wie weise ich das Standardverhalten bei Verwendung der ESC-/Enter-Taste zu?
- Wie werte ich die gewählte Taste aus?

Das folgende Beispiel soll diese Fragen beantworten.

BEISPIEL 21.29: *MessageDialog* mit Auswahl

C#

```
...
        async private void Button_Click_3(object sender, RoutedEventArgs e)
        {
```

Dialog erzeugen:

```
            MessageDialog msgdlg = new MessageDialog("Wählen Sie eine der Tasten",
                                                     "Die Qual der Wahl");
```

Wir definieren eigene Tasten (Beschriftung, Handler, CommandId):

```
            msgdlg.Commands.Add(new UICommand("Taste 1", null, "t1"));
            msgdlg.Commands.Add(new UICommand("Taste 2", null, "t2"));
```

Taste 1 mit der Enter-Taste verbinden:

```
            msgdlg.DefaultCommandIndex = 0;
```

Taste 2 mit der Escape-Taste verbinden:

```
            msgdlg.CancelCommandIndex = 1;
```

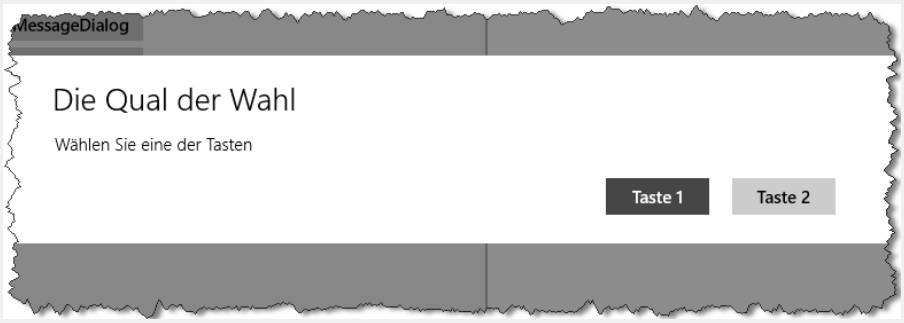
Dialoganzeige:

```
            var res = await msgdlg.ShowAsync();
```

Auswertung per CommandID:

```
            ListBox1.Items.Insert(0, "Gewählt: " + res.Id);
        }
...
```

Ergebnis



Nach dem Klick auf eine der Tasten wird entweder "t1" oder "t2" zurückgegeben.

Alternativen beim Auswerten der Tasten

Nicht jeder wird mit der im vorhergehenden Beispiel gezeigten Lösung glücklich, wer mag, kann die Auswertung der Tasten auch per Lambda-Ausdruck oder per Methode vornehmen. Zwei Beispiele zeigen, wie es geht:

BEISPIEL 21.30: Auswahl per Lambda-Ausdruck auswerten

C#

```
...
 async private void Button_Click_4(object sender, RoutedEventArgs e)
        {
            MessageDialog msgdlg = new MessageDialog("Wählen Sie eine der Tasten",
                                                     "Die Qual der Wahl");
```

Mit Hilfe des *Command*-Parameters können Sie im Lambda-Ausdruck die Beschriftung und optional die CommandId bestimmen:

```
            msgdlg.Commands.Add(new UICommand("Taste 1", (c) =>
            {
                ListBox1.Items.Insert(0, "Taste [" + c.Label + "] wurde gedrückt!");
            }));
            msgdlg.Commands.Add(new UICommand("Taste 2", (c) =>
            {
                ListBox1.Items.Insert(0, "Taste [" + c.Label + "] wurde gedrückt!");
            }));
            msgdlg.DefaultCommandIndex = 0;
            msgdlg.CancelCommandIndex = 1;
            var res = await msgdlg.ShowAsync();
            ListBox1.Items.Insert(0, "Gewählt: " + res.Id);
        }
...
```

HINWEIS: Beachten Sie, dass bei obigem Beispiel keine CommandId zurückgegeben wird, wir haben auch keine definiert.

BEISPIEL 21.31: Auswahl per Methode auswerten

C#

```
...
     async private void Button_Click_5(object sender, RoutedEventArgs e)
        {
            MessageDialog msgdlg = new MessageDialog("Wählen Sie eine der Tasten",
                                                     "Die Qual der Wahl");
```

Wir ordnen einen Methodenaufruf zu (mit CommandId):

```
            msgdlg.Commands.Add(new UICommand("Taste 1",
                  new UICommandInvokedHandler(this.CommandInvokedHandler), "MeineID"));
```

Hier ohne CommandId:

```
            msgdlg.Commands.Add(new UICommand("Taste 2",
                  new UICommandInvokedHandler(this.CommandInvokedHandler)));
```

BEISPIEL 21.31: Auswahl per Methode auswerten

```
            msgdlg.DefaultCommandIndex = 0;
            msgdlg.CancelCommandIndex = 1;
            var res = await msgdlg.ShowAsync();
            ListBox1.Items.Insert(0, "Gewählt: " + res.Id);
        }
```

Die Methode zum Auswerten der Tasten:

```
        private void CommandInvokedHandler(IUICommand command)
        {
            ListBox1.Items.Insert(0, "Taste [" + command.Label + "] wurde gedrückt!");
            ListBox1.Items.Insert(0, "ID ist [" + command.Id + "]");
        }
...
```

Ergänzungen

Doch wann sollen/dürfen Sie den *MessageDialog* eigentlich einsetzen?

- Grundsätzlich sollten Sie beachten, dass Ihre App vermutlich im Vordergrund steht, wenn der Dialog erscheint (andernfalls ist die App im Suspended-Modus). Eine Hinweismeldung aus dem "Untergrund" bekommen Sie damit also nicht hin.
- Nutzen Sie den Dialog nur für wichtige Entscheidungen, die einen direkten und sofortigen Nutzereingriff erfordern.
- Zeigen Sie nur wichtige Meldungen/Fehlermeldungen an, die für die weitere Programmausführung von Interesse sind.

HINWEIS: Für allgemeine Statusmeldungen etc. sollten Sie statt dem MessageDialog besser die so genannten Popup-Benachrichtigungen verwenden (siehe Abschnitt 21.3.2).

Ganz zum Schluss noch den einfachsten Aufruf der Methode:

```
        private void Button_Click_6(object sender, RoutedEventArgs e)
        {
            new MessageDialog("Hallo Welt!", "Mein Titel").ShowAsync();
        }
```

HINWEIS: Dieser Aufruf ist asynchron, was bei einem modalen "Fenster" aber meist nicht relevant ist.

21.3.2 Popup-Benachrichtigungen

Nicht jede Meldung ist es wert, dass der Nutzer gleich in hektische Betriebsamkeit verfällt und irgendwelche Tasten drücken muss, um z.B. zwangsweise einen *MessageDialog* zu schließen, der ihm mitteilt, dass die Nachrichtenliste eingelesen wurde.

Genau für diese Fälle wurden in Windows 8 die sogenannten Popup-Benachrichtigungen oder "Toast notifications" implementiert. Diese werden durch das **System** entweder sofort oder zeitgesteuert am rechten oberen Bildschirmrand temporär eingeblendet:

Popup-Benachrichtigungen können Texte und Bilder enthalten, die Definition erfolgt über ... tja, das ist dann auch schon das Problem ...

Wer schon mal einen Blick in das von Microsoft stammende Beispielprogramm

LINK: `http://code.msdn.microsoft.com/windowsapps/Toast-notifications-sample-52eeba29`

geworfen hat, wird sich sicher fragen, was sich die Entwickler dabei eigentlich gedacht haben. Als C#-Programmierer fühlen Sie sich hier etwas "angemeiert", erfolgt doch die Definition der Popup-Benachrichtigungen eigentlich in XML (nein nicht XAML). Sie haben jetzt drei Möglichkeiten:

- Sie schreiben den XML-Text selbst (als String),
- Sie verwenden die *NotificationExtensions*-Library von Microsoft[1]
- oder Sie verwenden bzw. erweitern unsere Hilfsklasse *ToastNotificationFactory*, die Ihnen die ganze Arbeit abnimmt.

Variante 1 und 2 werden Sie vermutlich nicht ganz zufriedenstellen, dies auch im Hinblick auf die Übersichtlichkeit und Länge des zu schreibenden Codes, für Spezialfälle kann es jedoch notwendig sein, darauf zurückzugreifen.

Den Quellcode der Klasse *ToastNotificationFactory* finden Sie im Praxisbeispiel

► 21.7.3 ToastNotifications einfach erzeugen

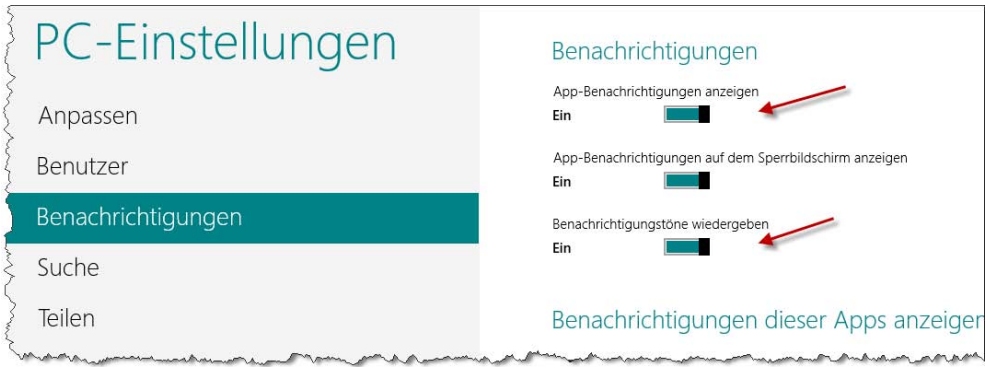
Doch bevor wir zur Verwendung unserer eigenen Klasse kommen, zunächst ein paar allgemeine Informationen zu den Popup-Benachrichtigungen.

Wann sollten Sie Popup-Benachrichtigungen einsetzen?

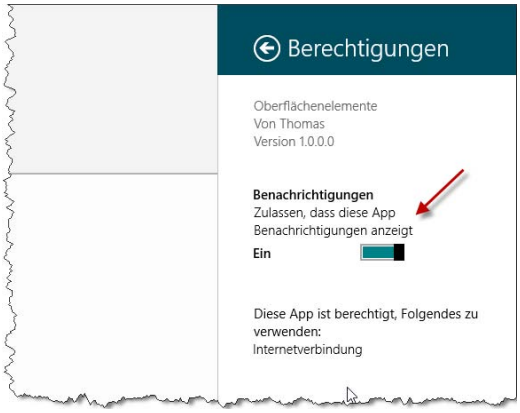
Gleich eine Warnung vorweg: Nein, nicht jedes Ereignis ist es wert, auch auf dem Nutzerbildschirm mit Signalton angezeigt zu werden. Machen Sie davon zu viel Gebrauch, kann es schnell passieren, dass sich der Nutzer "zur Wehr setzt". Damit ist nicht unbedingt die Deinstallation Ihrer App gemeint, nein, es geht auch einfacher.

[1] Siehe Quellcode obiges Beispiel.

Entweder per Einstellungen auf Systemebene:



oder individuell auf App-Ebene:



kann der Nutzer sich dieser Meldungen entledigen. Aus diesem Grund der Hinweis:

HINWEIS: Ihre App sollte auch bei abgeschalteten Popup-Benachrichtigungen ohne Einschränkungen funktionieren.

Wofür also einsetzen?

- Statusmitteilung über eingehende Nachrichten, Emails, SMS etc.
- Anzeige von Terminen und anderen vom Nutzer gewünschten Benachrichtigungen
- Abgeschlossene Updates und Hintergrund-Downloads
- Seitenwechsel innerhalb der App bei externen Ereignissen

Wofür nicht einsetzen?

- Für Fehlermeldungen etc. sollten Sie Popup-Benachrichtigungen nicht verwenden. Hier ist ein *MessageDialog* die bessere Wahl.
- Zu häufiges Anzeigen von Popup-Benachrichtigungen nervt mehr als es nutzt.
- Ist die App im Vordergrund, sollten Sie möglichst andere Oberflächenelemente zur Anzeige einsetzen (Flyouts, Popups etc.).

Was können Sie alles parametrieren?

Grundsätzlich müssen Sie zunächst zwischen zwei Arten von Popup-Benachrichtigungen unterscheiden:

- sofort aktivierte Popup-Benachrichtigungen (Klasse *ToastNotification*)
- zeitgesteuert angezeigte Popup-Benachrichtigungen (Klasse *ScheduledToastNotification*)

Letztere ermöglichen zusätzlich die Angabe eines Anzeigezeitpunktes.

HINWEIS: Da die Nachricht vom System angezeigt wird ist es vollkommen unerheblich, ob Ihre App läuft, sich im Suspended-Modus befindet oder ganz beendet ist. Ein Klick auf die geöffnete Meldung und Sie finden sich in der im Zweifel neu gestarteten App wieder.

Die Nachricht selbst kann mit verschiedenen Layouts angezeigt werden:

- einzeiliger Text
- einzeilige Überschrift und einzeiliger Text
- zweizeilige Überschrift und einzeiliger Text
- einzeilige Überschrift und zweizeiliger Text

Noch nicht genug? Alternativ können Sie bei allen obigen Varianten auch noch eine Grafik einblenden (das Symbol Ihrer App wird automatisch immer mit angezeigt). Nicht genug der Pein, auch die Anzeigedauer (kurz oder lang) lässt sich parametrieren, gleiches gilt für die Soundwiedergabe (es geht auch ohne Töne).

HINWEIS: Die angezeigten Bilder sollten kleiner als 200 KByte und kleiner als 1024x1024 Pixel sein, andernfalls sehen Sie ... nichts.

Ganz nebenbei müssen/können Sie auch auf den Klick reagieren. Wird die Benachrichtigung ohne Klick geschlossen, werden Sie ebenfalls durch ein Ereignis darauf aufmerksam gemacht.

Und wie funktioniert das alles?

Prinzipiell müssen Sie, wie auch immer, den erforderlichen XML-Code für eine derartige Nachricht zusammenstellen und mit dem Umweg über ein *XMLDocument* damit eine Instanz der Klasse

ToastNotification bzw. *ScheduledToastNotification* erzeugen. Anschließend können Sie über den *ToastNotificationManager* diese Nachricht zur direkten oder zeitgesteuerten Anzeige bringen.

BEISPIEL 21.32: XML-Code für eine Nachricht

XML

Eine Nachricht mit Kopfzeile, zwei Textzeilen sowie einem Bild:

```
<toast launch="Mein eigener Parameter">
  <visual version="1">
    <binding template="toastImageAndText04">
      <text id="1">ToastNotification</text>
      <text id="2">Mit Zeitsteuerung</text>
      <text id="3">Hier klicken!</text>
      <image id="1" src="bulbon.png" alt="Bild" />
    </binding>
  </visual>
</toast>
```

Wem das zu kompliziert ist, der kann sich die Klasse *ToastNotificationFactory* aus unserem Beispielprogramm kopieren und in seine Projekte einfügen.

Unser ToastNotification-"Bäckerei"

Den Quellcode und die komplette Beschreibung der Klasse *ToastNotificationFactory* finden Sie im Praxisbeispiel

▶ 21.7.3 ToastNotifications einfach erzeugen

An dieser Stelle wollen wir uns nur mit der Verwendung anhand einiger Beispiele beschäftigen.

BEISPIEL 21.33: Einfache Meldung mit Kopfzeile und einer Textzeile anzeigen

C#

```
...
        private void Button_Click_7(object sender, RoutedEventArgs e)
        {
            DOKO.ToastNotificationFactory tf = new DOKO.ToastNotificationFactory(
                        "Meine Kopfzeile", "Meine wichtige Meldung an den User");
            tf.Show();
        }
...
```

Ergebnis

Es geht auch noch kürzer:

```
new DOKO.ToastNotificationFactory(
       "Meine Kopfzeile", "Meine wichtige Meldung an den User").Show()
```

Ein Klick auf die Popup-Benachrichtigung zeigt im obigen Beispiel keine Reaktion. Dies wollen wir im nächsten Beispiel ändern.

BEISPIEL 21.34: Reaktionen auf Ereignisse implementieren

C#

In der *BasicPage1.xaml.cs*:

```
...
        private void Button_Click_8(object sender, RoutedEventArgs e)
        {
```

Wir erzeugen eine dreizeilige Nachricht, die einen Parameter enthält:

```
            DOKO.ToastNotificationFactory tf = new DOKO.ToastNotificationFactory(
               "Meine Kopfzeile", "Hier wird auch ein Parameter übergeben", "Hier klicken!");
            tf.Parameter = "Mein eigener Parameter";
```

Wir klinken uns in zwei Ereignisse ein:

```
            ToastNotification toast = tf.CreateToast();
            toast.Activated += toast_Activated;
            toast.Dismissed += toast_Dismissed;
```

Meldung über den *ToastNotificationManager* anzeigen:

```
            ToastNotificationManager.CreateToastNotifier().Show(toast);
        }
...
```

Dieses Ereignis wird ausgelöst, wenn die Popup-Benachrichtigung **nicht** angeklickt wird:

```
        async void toast_Dismissed(ToastNotification sender, ToastDismissedEventArgs args)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                ListBox1.Items.Add("Toast Dismissed: " + args.Reason.ToString());
            });
        }
```

Dieses Ereignis wird ausgelöst, wenn die Popup-Benachrichtigung durch den Nutzer angeklickt wurde:

```
        async void toast_Activated(ToastNotification sender, object args)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, () =>
            {
                ListBox1.Items.Add("Toast Activated");
            });
        }
```

BEISPIEL 21.34: Reaktionen auf Ereignisse implementieren

Wird ein Parameter übergeben, rufen wir aus der *App.xaml.cs* diese Methode auf:

```
        public void Parameterübergabe(String args)
        {
            ListBox1.Items.Add("Toast : " + args);
        }
...
```

In der *App.xaml.cs*:

```
...
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
            Frame rootFrame = Window.Current.Content as Frame;
            if (rootFrame == null)
            {
                rootFrame = new Frame();
                if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
                {
                    //TODO: Load state from previously suspended application
                }
                Window.Current.Content = rootFrame;
            }

            if (rootFrame.Content == null)
            {
                if (!rootFrame.Navigate(typeof(BasicPage1), args.Arguments))
                    throw new Exception("Failed to create initial page");
            }
            else
            {
```

Sollte die Nachricht einen Parameter aufweisen, ruft das System die *OnLaunched*-Methode auf und übergibt den Parameter an die App. Wir geben diesen einfach per Methode *Parameterübergabe* an die App weiter:

```
                BasicPage1 page1 = (BasicPage1)rootFrame.Content;
                page1.Parameterübergabe(args.Arguments);
            }
            Window.Current.Activate();
        }
...
```

Führen Sie obiges Beispiel aus so sehen Sie, dass bedingt durch die Parameterübergabe die App erneut die *OnLaunched*-Methode durchlaufen wird. Lassen Sie den Parameter weg, tritt nur das *ToastNotification*-Ereignis *Activated* auf, *OnLaunched* wird nicht aufgerufen.

Unser nächstes Beispiel zeigt eine Benachrichtigung mit Bildanzeige.

BEISPIEL 21.35: *ToastNotification* mit Bild

C#

```
...
        private void Button_Click_9(object sender, RoutedEventArgs e)
        {
            DOKO.ToastNotificationFactory tf = new DOKO.ToastNotificationFactory(
                        "Meine Kopfzeile", "Hier wird ein Bild angezeigt!");
```

Bildauswahl:

```
            tf.Bild = "bulbon.png";
```

Die Nachricht 25 statt 7 Sekunden anzeigen:

```
            tf.LongRunning = true;
```

Keinen Hinweiston wiedergeben:

```
            tf.Silent = true;
            tf.Show();
        }
...
```

Ergebnis

Last but not least, wollen wir Ihnen auch noch eine zeitgesteuerte Variante demonstrieren:

BEISPIEL 21.36: Zeitgesteuerte Anzeige der Benachrichtigung

C#

```
...
        private void Button_Click_10(object sender, RoutedEventArgs e)
        {
            DOKO.ToastNotificationFactory tf = new
                                    DOKO.ToastNotificationFactory("ToastNotification",
                                    "Mit Zeitsteuerung", "Hier klicken!");
            tf.Bild = "bulbon.png";
            tf.Parameter = "Mein eigener Parameter";
```

Verzögerung um eine Minute:

```
            tf.AddToSchedule(DateTime.Now.AddMinutes(1));
        }
...
```

Rufen Sie die Funktion auf und beenden Sie die App bzw. wechseln Sie zu einer anderen App. Nach einer Minute taucht die Meldung am rechten oberen Rand auf, klicken Sie darauf, wird Ihre App gestartet oder in den Vordergrund gebracht.

Prüfen auf Funktionsfähigkeit

Wir hatten es ja schon erwähnt, der Nutzer hat mehrere Möglichkeiten, die Popup-Benachrichtigungen zu deaktivieren. Wollen Sie in Ihrer App ermitteln ob dies der Fall ist, können Sie folgende Eigenschaft abfragen:

```
ToastNotificationManager.CreateToastNotifier().Setting
```

Mögliche Rückgabewerte sind:

- *Enabled*
- *DisabledForApplication*
- *DisabledForUser*
- *DisabledByGroupPolicy*
- *DisabledByManifest*

Damit wollen wir es an dieser Stelle belassen, wenden wir uns weiteren interessanten Elementen der Oberfläche zu.

21.3.3 PopUp/Flyouts

Wer hat sie noch nicht vermisst? Dialoge, die über der eigentlichen Seite angezeigt werden, um zum Beispiel Detaildaten einzugeben oder Informationen anzuzeigen. Einzig sinnvolle Alternative in der schönen neuen bunten WinRT-Welt ist das *Popup*-Control, das sich mit etwas Aufwand auch zum Flyout "umrüsten" lässt.

Grundsätzlich stehen Ihnen zwei Ansätze zur Verfügung, wie Sie zu einem Popup kommen:

- per XAML-Code
  Fügen Sie dazu das Control zum Beispiel im Content eines *Grid*-Controls ein. Im Normalfall ist das Popup nicht sichtbar. Beim Anzeigen wird dass *PopUp*-Control dann am Parent ausgerichtet. Einen Offset können Sie per *HorizontalOffset* und *VerticalOffset* festlegen.
- per C#-Code
  Hier ist es günstig, wenn Sie den Content des Popup-Controls zum Beispiel als UserControl erstellen (XAML-Code) und dann als Child zuweisen. Andernfalls haben Sie schnell sehr viel Code produziert, nur um eine Schaltfläche und ein paar andere Controls anzuzeigen.

BEISPIEL 21.37: PopUp per XAML-Code erzeugen

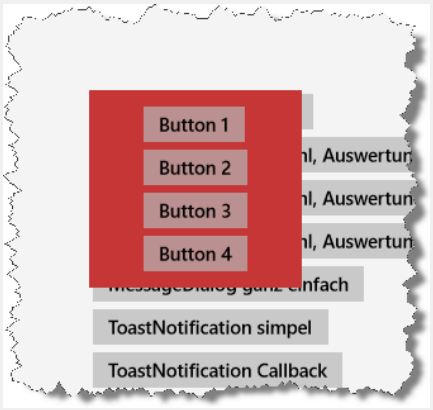
XAML

```
...
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition></ColumnDefinition>
                    <ColumnDefinition></ColumnDefinition>
                </Grid.ColumnDefinitions>
...
                <Popup x:Name="Popup1" IsLightDismissEnabled="True" >
```

BEISPIEL 21.37: PopUp per XAML-Code erzeugen

```
                    <Border Padding="40,10,40,10" Background="#FFC73636">
                        <StackPanel>
                            <Button>Button 1</Button>
                            <Button>Button 2</Button>
                            <Button>Button 3</Button>
                            <Button>Button 4</Button>
                        </StackPanel>
                    </Border>
                </Popup>
            </Grid>
...
```

Ergebnis



Etwas umständlicher wird es per C#-Code gerade dann, wenn Sie die enthaltenen Controls parametrieren wollen:

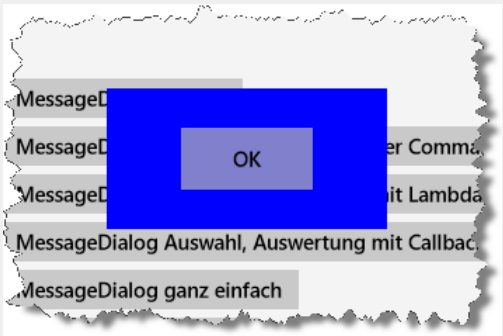
BEISPIEL 21.38: PopUp per C#-Code erzeugen

C#

```
...
        private void Button_Click_13(object sender, RoutedEventArgs e)
        {
            Popup popup = new Popup();
            popup.Width = 200;
            popup.Height = 150;
            popup.VerticalOffset = 200;
            popup.HorizontalOffset = 200;
            Border bord = new Border() { Width = 200, Height = 100,
                                         Background = new SolidColorBrush(Colors.Blue) };
            bord.Child = new Button() { Width = 100, Height = 50, Content = "OK" };
            popup.Child = bord;
            popup.IsLightDismissEnabled = true;
            popup.IsOpen = true;
        }
...
```

BEISPIEL 21.38: PopUp per C#-Code erzeugen

Ergebnis



Wie Sie das Popup anzeigen, haben Sie ja im obigen Beispiel bereits gesehen, Sie setzen einfach die Eigenschaft *IsOpen* auf *True*.

Das Grundverhalten zum Schließen des Popups steuern Sie mit der Eigenschaft *IsLightDismissEnabled*. Diese entscheidet darüber, ob das Popup ausgeblendet wird wenn der Nutzer außerhalb klickt (*True*) oder ob es geöffnet bleibt (*False*). Im letzteren Fall müssen Sie dann das Popup per Code (*IsOpen*) wieder schließen.

Ein leidiges Thema ist die Ausrichtung des *PopUp*-Controls. Eine Eigenschaft *Placement* wie in WPF werden Sie nicht finden, Sie müssen schon mit *HorizontalOffset* und *VerticalOffset* vorlieb nehmen. Wem diese Rechnerei zu kompliziert ist, der kann folgende Routine[1] verwenden:

BEISPIEL 21.39: UserControls im Popup einblenden und positionieren

C#

```
...
        public static Popup ShowPopup(FrameworkElement source, UserControl control,
                                bool IsLightDismissEnabled = true)
        {
            Popup flyout = new Popup();
            flyout.IsLightDismissEnabled = IsLightDismissEnabled;
            GeneralTransform gt = source.TransformToVisual(Window.Current.Content);
            var absolutePosition = gt.TransformPoint(new Point(0, 0));
            control.Measure(new Size(Double.PositiveInfinity, double.PositiveInfinity));
            flyout.VerticalOffset = absolutePosition.Y - control.ActualHeight - 10;
            flyout.HorizontalOffset = (absolutePosition.X + source.ActualWidth / 2) -
                                      control.ActualWidth / 2;
            flyout.Child = control;
            var transitions = new TransitionCollection();
            transitions.Add(new PopupThemeTransition() { FromHorizontalOffset = 0,
                                                         FromVerticalOffset = 100 });
            flyout.ChildTransitions = transitions;
            flyout.IsOpen = true;
            return flyout;
```

[1] Quelle: *http //blogs.msdn.com/b/eternalcoding/archive/2012/07/03/tips-and-tricks-for-c-metro-developers-the-flyout-control.aspx*

BEISPIEL 21.39: UserControls im Popup einblenden und positionieren

```
        }
...
```

Die Verwendung:

```
        private void Button_Click_12(object sender, RoutedEventArgs e)
        {
            ShowPopup(Button12, new MyUserControl1());
        }
...
```

Der obigen Funktion übergeben Sie ein Bezugs-Control (z.B. eine Schaltfläche) und ein *UserControl* (das ist der Inhalt des neuen Popups). Optional können Sie den Wert der Eigenschaft *IsLightDismissEnabled* bestimmen. Die Funktion erzeugt daraufhin ein Popup, blendet dieses an der richtigen Position mit einer Animation ein und gibt ein *Popup*-Objekt zurück (z.B. zum aktiven Ausblenden per *IsOpen*).

Bleibt die Frage nach den "echten" Flyouts. Ab Werk werden Sie diese nicht vorfinden, Alternativen finden sich entweder hier

LINK: `https://github.com/timheuer/callisto`

oder hier

LINK: `http://code.msdn.microsoft.com/windowsapps/CharmFlyout-A-Metro-Flyout-25fe53b6`

HINWEIS: Auf die Verwendung dieser Controls gehen wir aus naheliegenden Gründen an dieser Stelle nicht ein.

Sie können allerdings ein *Popup* problemlos mit einer Animation versehen, wie es auch in Beispiel 21.39 passiert. Möchten Sie beispielsweise einen Effekt wie beim Charmbar realisieren, nutzen Sie folgenden Code:

BEISPIEL 21.40: Ein *Popup* wie den Charmbar ein-/ausblenden

XAML

```
...
        <Popup x:Name="Popup2" IsLightDismissEnabled="True">
            <Popup.ChildTransitions>
                <TransitionCollection>
                    <PaneThemeTransition />
                </TransitionCollection>
            </Popup.ChildTransitions>
            <Border Background="Blue" Width="400" Height="200">
                <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
                    <TextBlock>Hallo</TextBlock>
                    <Button Click="Button_Click_15">OK</Button>
                </StackPanel>
            </Border>
```

BEISPIEL 21.40: Ein *Popup* wie den Charmbar ein-/ausblenden

```
        </Popup>
...
```

C# Beim Einblenden müssen wir das Popup am rechten Rand ausrichten:

```
private void Button_Click_14(object sender, RoutedEventArgs e)
{
    Popup2.HorizontalOffset = Window.Current.Bounds.Width - 400;
    Popup2.IsOpen = true;
}
```

21.3.4 Das PopupMenu einsetzen

Ein naher, etwas spezialisierter Verwandter des Popups ist das *PopupMenu*. Diesem können Sie einzelne *UICommand*-Objekte hinzufügen (siehe auch 21.3.1, Seite 1086), die wiederum eine Beschriftung (*Label*) und eine Id aufweisen können. Eingeblendet wird das *PopupMenu* per *ShowAsync*- oder *ShowForSelectionAsync*-Methode. Hier wird es dann aber auch etwas diffizil, beide Methoden erwarten eine Positionsangabe. Leider ist dies mit Rechnerei und einem Haufen Quellcode verbunden. Sie finden deshalb in unserem Beispiel eine Alternative in Form einer Erweiterungsmethode, der Sie einfach das aufrufende Control (*Button*, *TextBox* etc.) übergeben, die Position des *PopupMenue* wird dann intern berechnet.

BEISPIEL 21.41: Verwendung *PopupMenu*

C#

```
...
using Windows.UI.Popups;
...
```

Mit dem Klick auf eine Schaltfläche erstellen wir das *PopupMenu* und zeigen es an:

```
async private void Button_Click_17(object sender, RoutedEventArgs e)
{
    var popmenu = new PopupMenu();
```

Der erste Eintrag nutzt eine Lambda-Funktion als Reaktion auf den Klick:

```
    popmenu.Commands.Add(new UICommand("Erster Eintrag", (command) =>
    {
        ListBox1.Items.Add(command.Label);
    }));
```

Die folgenden Einträge sind mit einem extra Handler verknüpft:

```
    popmenu.Commands.Add(new UICommand("Zweiter Eintrag",
                         new UICommandInvokedHandler(test), "2"));
    popmenu.Commands.Add(new UICommand("Dritter Eintrag",
                         new UICommandInvokedHandler(test), "dritter"));
    popmenu.Commands.Add(new UICommand("Vierter Eintrag",
                         new UICommandInvokedHandler(test), "vierter"));
```

BEISPIEL 21.41: Verwendung *PopupMenu*

Aufrufvariante 1:

```
GeneralTransform transform = (sender as Button).TransformToVisual(this);
Point point = transform.TransformPoint(new Point(100,150));
await popmenu.ShowAsync(point);
```

Aufrufvariante 2:

```
GeneralTransform transform = (sender as Button).TransformToVisual(this);
Point point = transform.TransformPoint(new Point(100,150));
await popmenu.ShowForSelectionAsync(new Rect(point,
                new Size((sender as Button).ActualWidth,
                (sender as Button).ActualHeight)));
```

Aufrufvariante 3 (unsere Erweiterungsmethode):

```
await popmenu.ShowAsync(sender as Button);
}
```

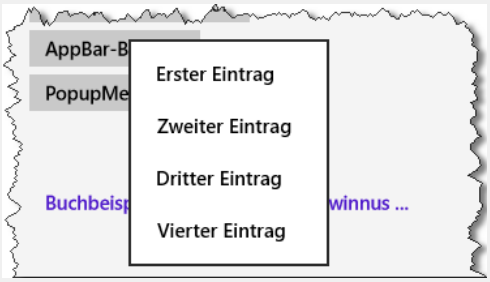
Welche Aufrufvariante intuitiver ist, dürfte schnell erkennbar sein.

C#

Mit der folgenden Erweiterungsmethode wird das *PopupMenu* an einem Control ausgerichtet und angezeigt:

```
...
    static class Erweiterungsmethoden
    {
        public async static Task<IUICommand> ShowAsync(this PopupMenu menu,
                                                FrameworkElement control)
        {
            GeneralTransform transform = control.TransformToVisual(Window.Current.Content
                                                                as Frame);
            Point point = transform.TransformPoint(new Point(100, 150));
            return await menu.ShowForSelectionAsync(new Rect(point,
                        new Size(control.ActualWidth, control.ActualHeight)));
        }
    }
...
```

Ergebnis



21.3.5 Eine AppBar verwenden

Wir hatten uns ja bereits im Rahmen der Oberflächengestaltung (Abschnitt 18.1.3, Seite 902) erstmals mit dem *AppBar*-Control beschäftigt, an dieser Stelle soll es jetzt um die praktische Umsetzung gehen. Die Definition erfolgt erwartungsgemäß per XAML, wobei Sie hier zwischen den beiden Anzeigezielen

- *<Page.TopAppBar>* und
- *<Page.BottomAppBar>*

unterscheiden müssen. Eine *AppBar*, die Sie am oberen Rand der Seite anzeigen, dient der Navigation innerhalb Ihrer Anwendung, die untere *AppBar* ist der Anzeige von Befehlen und Werkzeugen vorbehalten.

Eine *AppBar* ist standardmäßig zunächst nicht zu sehen, Sie können diese per

- Maus (rechte Maustaste),
- Wischgeste von oben oder unten,
- Tastatur (Windowstaste+Z)
- oder per Eigenschaft *IsOpen* (*True*)

einblenden.

HINWEIS: Sind eine obere und eine untere *AppBar* definiert, werden auch immer beide zugleich angezeigt, Ausnahme ist das Einblenden per *IsOpen*-Eigenschaft..

Die *AppBar* wird im Normalfall automatisch ausgeblendet, wenn der Nutzer außerhalb der *AppBar* klickt. Dieses Verhalten lässt sich jedoch verändern, indem Sie die Eigenschaft *IsSticky* auf *True* setzen. Zum Schließen der *AppBar* ist jetzt eine der obigen Aktionen bzw. das Setzen der Eigenschaft *IsOpen* nötig. In diesem Zusammenhang sollten Sie auch einen Blick auf die beiden Ereignisse *Opened* und *Closed* werfen, hier können Sie gegebenenfalls Anpassungen an der *AppBar* vornehmen (kontextabhängige Befehle, Navigationsoptionen etc.).

Innerhalb der *AppBar* können Sie Schaltflächen für das Ausführen von Routinen bzw. das Setzen von Eigenschaften platzieren. Orientieren Sie sich dabei an den Layoutvorgaben von Microsoft, d.h., verteilen Sie die Controls möglichst am linken oder rechten Rand, da diese Bereiche bei einer Toucheingabe optimal erreichbar sind.

Bevor Sie sich jetzt intensiv mit dem Erstellen von Schaltflächen für die untere Befehlsleiste beschäftigen, sollten Sie einmal einen Blick in die Liste der vordefinierten Button-Layouts werfen. Nebenstehende Abbildung zeigt eine Übersicht der möglichen Styles.

HINWEIS: Wie Sie der folgenden Abbildung entnehmen können, sind alle Schaltflächen englisch beschriftet. Möchten Sie dies ändern, können Sie entweder die Datei *StandardStyles.xaml* editieren oder Sie setzen die Eigenschaft *AutomationProperties.Name* für die jeweilige Schaltfläche.

Accounts
Add
Add Friend
Admin
Align Center
Align Left
Align Right
All Apps
Attach
Attach Camera
Audio
Back to Window
BlockContact
Bold
Bookmarks
Browse Photos
Bullets
Calculator
Calendar
Day
Calendar Reply
Week
Caption
Cc
Cellphone
Characters
Clear Selection
Clock
Close Pane
Comment
Contact
Contact
Contact Info
Presence
Copy
Crop
Cut
Delete
Directions
Disable Updates
Discard
Disconnect
Dislike
Dock Bottom
Dock Left
Dock Right
Document
Download
Edit
Emoji2
Emoji
Favorite
Filter
Flag
Folder
Font
Font Color
Decrease Font
Increase Font
Font Size
Four Bars
Full Screen
Globe
Go
Go to Start
Today
Hang Up
Help
Hide Bcc
Highlight
Home
Import All
Important
Import
Italic
Keyboard
Upload
Library
Like
Like/Dislike
Link
List
Mail2
Mail
Forward Mail
Reply All
Reply
Manage
Map
Map Drive
Map Pin
Memo
Message
Microphone
More
Move to Folder
Music Info
Mute
New Folder
New Window
Next
No
One Bar
Open File
Open Loal
Open Pane
Open With
Orientation
Other User
Out
Outline Star
Page2
Page
Paste
Pause
People
Permisions
Phone
Phonebook
Photo
Pictures
Pin
Placeholder
Play
Post Update
Preview Link
Previous
Priority
Protected
Read
Redo
Refresh
Remote
Remove
Rename
Repair
Repeat All
Repeat Once
Report Hacked
Reshare
Rotate
Rotate Camera
Save
Save Local
Search
Select All
Send
Set Lockscreen
Settings
Set Title
Shop
Show Bcc
Show Results
Shuffle
Skip Ahead
Skip Back
Slideshow
Solid Star
Sort
Stop
Stop Slideshow
Street
Switch
Switch Apps
Sync
Sync Folder
Tag
Target
Three Bars
Touch Pointer
Trim
Two Bars
Two Page
Underline
Undo
Unfavorite
Unpin
Unsync Folder
Upload
Skydrive
Video
Video Chat
View All
View
Volume
Webcam
World
Yes
Zero Bars
Zoom
Zoom In
Zoom Out

BEISPIEL 21.42: *AppBar* verwenden

XAML

```
...
<common:LayoutAwarePage
...
    mc:Ignorable="d">
```

Wir definieren zuerst die obere Navigationsleiste:

```
    <Page.TopAppBar>
```

Diese Liste wird beim Öffnen der Seite bereits eingeblendet (*IsOpen=True*):

```
        <AppBar Name="TopAppBar1" Background="Lavender" IsOpen="True">
```

Das folgende *Grid* und die enthaltenen *StackPanel*s ermöglichen die Ausrichtung der Inhalte am linken bzw. rechten Rand der Seite:

```
            <Grid Margin="10,0,10,0">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="50*"/>
                    <ColumnDefinition Width="50*"/>
                </Grid.ColumnDefinitions>
                <StackPanel Orientation="Horizontal" Grid.Column="0"
                            HorizontalAlignment="Left">
```

Die eigentlichen Schaltflächen (Eigenkonstruktion):

```
                    <Button>
                        <StackPanel HorizontalAlignment="Center">
                            <Image Source="bulbon.png"/>
                            <TextBlock>Rubrik A</TextBlock>
                        </StackPanel>
                    </Button>
                    <Button>
                        <StackPanel HorizontalAlignment="Center">
                            <Image Source="bulboff.png"/>
                            <TextBlock>Rubrik B</TextBlock>
                        </StackPanel>
                    </Button>
                </StackPanel>
            </Grid>
        </AppBar>
    </Page.TopAppBar>
```

Die Befehlsleiste am unteren Rand ist zunächst ausgeblendet, die enthaltenen Schaltflächen haben wir aus der Liste der verfügbaren Styles ausgewählt:

```
    <Page.BottomAppBar>
        <AppBar Name="BottomAppBar1" Background="Lavender" >
            <Grid Margin="10,0,10,0">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="50*"/>
```

BEISPIEL 21.42: *AppBar* verwenden

```
                <ColumnDefinition Width="50*"/>
            </Grid.ColumnDefinitions>
            <StackPanel Orientation="Horizontal" Grid.Column="0"
                        HorizontalAlignment="Left">
```

Die "Eindeutschung" nehmen wir über die Eigenschaft *AutomationProperties.Name* vor:

```
                <Button Style="{StaticResource EditAppBarButtonStyle}"
                        Click="Button_Click_3" AutomationProperties.Name="Editieren" />
                <Button Style="{StaticResource SaveAppBarButtonStyle}"
                        Click="Button_Click_4" AutomationProperties.Name="Sichern"/>
                <Button Style="{StaticResource DeleteAppBarButtonStyle}"
                        AutomationProperties.Name="Löschen"/>
            </StackPanel>
            <StackPanel Orientation="Horizontal" Grid.Column="1" >
...
            </StackPanel>
        </Grid>
    </AppBar>
</Page.BottomAppBar>
...
```

C#

```
...
```

Beim Klick auf die "Editieren"-Schaltfläche wird ein *Popup* geöffnet, Inhalt ist ein *UserControl* (quasi der Ersatz für einen Dialog):

```
private void Button_Click_3(object sender, RoutedEventArgs e)
 {
     Popup flyout = new Popup();
     flyout.IsLightDismissEnabled = true;
     UserControl uc = new MyUserControl1();
     flyout.Child = uc;
```

Wir richten den Dialog an der linken Kante des Buttons aus, die Höhe müssen wir leider direkt setzen:

```
     GeneralTransform gt = (sender as Button).TransformToVisual(
                                          Window.Current.Content);
     var absolutePosition = gt.TransformPoint(new Point(0, 0));
     uc.Measure(new Size(Double.PositiveInfinity, double.PositiveInfinity));
     flyout.VerticalOffset = absolutePosition.Y - 350;
     flyout.HorizontalOffset = absolutePosition.X;
     var transitions = new TransitionCollection();
     transitions.Add(new PopupThemeTransition() { FromHorizontalOffset = 0,
                                                  FromVerticalOffset = 100 });
     flyout.ChildTransitions = transitions;
     flyout.IsOpen = true;
 }
```

BEISPIEL 21.42: *AppBar* verwenden

Der Klick auf die Schaltfläche "Sichern" zaubert ein *PopupMenu* hervor:

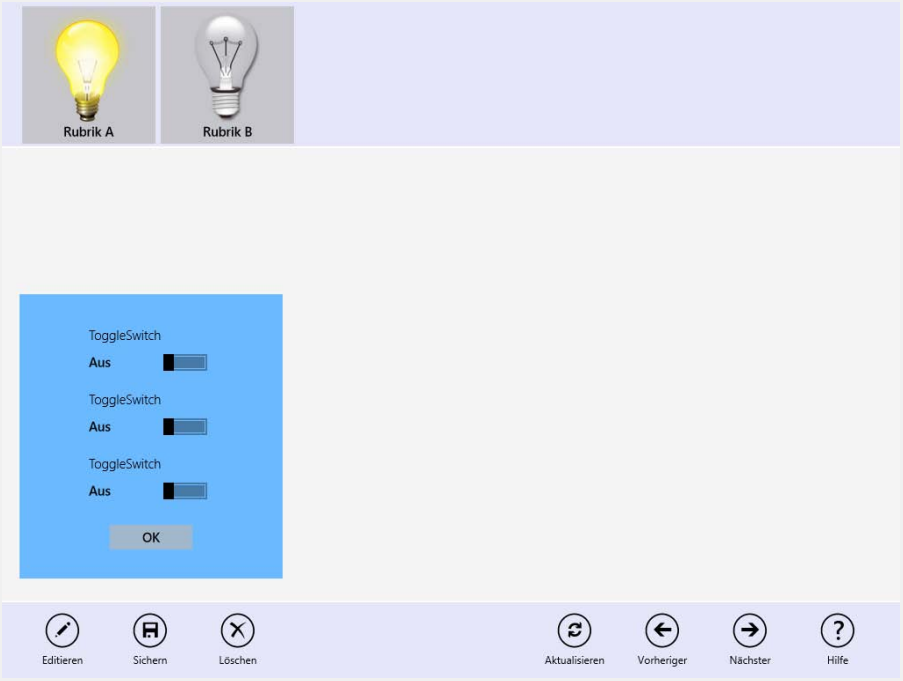
```
async private void Button_Click_4(object sender, RoutedEventArgs e)
{
    var popupMenu = new PopupMenu();
    popupMenu.Commands.Add(new UICommand("Eintrag 1"));
    popupMenu.Commands.Add(new UICommand("Eintrag 2"));
    popupMenu.Commands.Add(new UICommand("Eintrag 4"));
    popupMenu.Commands.Add(new UICommand("Eintrag 5"));
```

Hier nutzen wir wieder unsere Erweiterungsmethode zur Anzeige:

```
    await popupMenu.ShowAsync(sender as Button);
}
...
```

Ergebnis

Unsere App in Aktion:



HINWEIS: Beim Öffnen der Seite ist zunächst nur der obere *AppBar* zu sehen. Klicken Sie mit der rechten Maustaste auf die Seite, wird auch noch der untere *AppBar* angezeigt, ein erneuter Klick blendet beide *AppBar*s wieder aus.

Damit schließen wir diesen Abschnitt und wenden uns einem leider von Microsoft recht stiefmütterlich behandelten Thema zu.

21.4 Datenbanken und Windows Store Apps

Oh, da scheinen die Entwickler mal wieder eine "Kleinigkeit" vergessen zu haben. Prinzipiell steht Ihnen derzeit "ab Werk" **keine Datenbankengine** für Ihre App zur Verfügung! Das dürfte zunächst ein kompletter Show-Stopper für viele Anwendungen sein, die derzeit noch um eine lokale Desktop-Datenbank herum aufgebaut sind. Schnelle und vor allem einfache Abhilfe ist hier nicht in Sicht. Mittlerweile scheint aber Microsoft zur Einsicht gelangt zu sein, dass hier eine gewaltige Lücke klafft.

21.4.1 Der Retter in der Not: SQLite!

Auf der Website

LINK: http://timheuer.com/blog/archive/2012/06/05/howto-video-using-sqlite-in-metro-style-app.aspx

wird eine Lösung vorgestellt, die allerdings nicht von Microsoft stammt.

Wir hatten uns in unserem letzten Datenbankbuch bereits ausführlicher[1] damit beschäftigt: Es geht um SQLite, das mittlerweile auch in einer für WinRT-tauglichen Form verfügbar ist.

21.4.2 Verwendung/Kurzüberblick

Dass wir Ihnen an dieser Stelle keinen kompletten Überblick zu SQLite bieten können, verzeihen Sie uns bitte (wir mussten aus Platzgründen schon recht viele Kapitel in das E-Book auslagern) und so wollen wir uns auf einige wichtige Punkte beschränken[2].

Was eigentlich ist SQLite?

Bei SQLite handelt es sich um eine Desktop-Datenbankengine, die im Gegensatz zu einem SQL Server ohne eine extra Server-Anwendung auskommt. Die komplette Funktionalität wird von einer DLL bereitgestellt, die Anwendung greift also direkt auf den eigentlichen Datenspeicher zu. Der Clou an dieser Lösung: Sie können trotz allem mit SQL als Abfragesprache arbeiten, müssen sich also nicht lange an eine neue Schnittstelle gewöhnen[3].

Einen Überblick zum Datenformat, zur verwendeten SQL-Syntax und zur DLL-Schnittstelle bietet Ihnen die folgende Website:

LINK: http://www.sqlite.org/

Im Folgenden wollen wir mit Ihnen zunächst die Vor- und Nachteile einer Entscheidung für SQLite diskutieren.

---

[1] Unter diesen Umständen werden wir das entsprechende Kapitel in unserem Datenbankbuch wohl noch kräftig ausbauen müssen.

[2] Wesentlich mehr zu SQLite und dem passenden C#-Wrapper für diese Datenbank-Engine finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012] aus dem Microsoft Press-Verlag.

[3] Am besten können Sie SQLite noch mit dem SQL Server Compact vergleichen, beide haben einen konzeptionell ähnlichen Ansatz.

Vorteile

Davon bietet SQLite jede Menge:

- Die Datenbankengine ist winzig im Vergleich zu den etablierten Produkten (die DLL hat lediglich eine Größe von ungefähr 880 KByte).
- Es ist keinerlei administrativer Aufwand notwendig, wenn Sie mal vom Speichern der eigentlichen Datendatei absehen.
- Das Format ist ideal für die Verwendung im Zusammenhang mit WinRT, da geringer Ressourcenbedarf.
- Alle Daten sind in einer Datei zusammengefasst.
- Die komplette Engine befindet sich in einer bzw. in zwei Dateien.
- SQLite implementiert einen Großteil der SQL92-Spezifikation.
- SQLite-Datenbanken sind plattformkompatibel, d h., Sie können die Datei problemlos mit anderen Systemen auslesen und bearbeiten (z.B. PDA). Für fast jede Plattform und Programmiersprache werden entsprechende Schnittstellen angeboten.
- Im Gegensatz zum SQL Server Compact kann man bei einer maximalen Datenbankgröße von 2 Terabyte kaum noch von einer Größenbegrenzung sprechen. Hier fallen die Restriktionen von WinRT in Bezug auf die maximale Größe das App-Datenverzeichnisses mehr ins Gewicht.
- Datenbanken können verschlüsselt werden.
- Unterstützung für Trigger, Views und Constraints.
- SQLite ist komplett kostenlos, der Quellcode ist ebenfalls frei verfügbar.

Nachteile

Jede Medaille hat zwei Seiten und so müssen Sie auch bei SQLite mit einigen Einschränkungen und Nachteilen leben.

- Grundsätzlich sollten Sie immer das Konzept als Desktop-Datenbank im Auge behalten. Sie können zwar mit mehreren Anwendungen auf die Datendatei zugreifen, allerdings ist der Schreibmechanismus der Engine etwas eigenwillig, nur ein Prozess kann exklusiv auf die Datenbank zugreifen, Lesezugriffe werden in dieser Zeit geblockt. Im Zusammenhang mit Apps stellt dies jedoch kein Problem dar.
- Keine Unterstützung für Stored Procedures und UDFs, Sie können jedoch eigene Scalar- und Aggregat-Funktionen schreiben.
- Keine geschachtelten Transaktionen.
- Keine Replikationsunterstützung, Sie können jedoch eine zweite Datenbank mit ATTACH einbinden und nachfolgend die Daten mit einer Abfrage über die betreffenden Tabellen synchronisieren.

- Keine Unterstützung für Nutzer- und Rechteverwaltung, es handelt sich um eine Desktop-Datenbank, die Sie jedoch verschlüsseln können.

Für alle, die sich jetzt voller Euphorie auf die SQLite-Entwicklung stürzen wollen, gleich ein beachtlicher Dämpfer:

HINWEIS: In WinRT steht Ihnen ADO.NET nicht zur Verfügung. Also keine DataSets etc., alle Zugriffe erfolgen über SQLite-spezifische Methoden, der Portierungsaufwand einer bisherigen ADO.NET-Anwendung sollte also nicht unterschätzt werden!

21.4.3 Installation

Mit der SQLite-DLL allein ist es nicht getan, wir benötigen zusätzlich noch einen Wrapper, der uns die Library-Funktionen in ein halbwegs nutzbares Format überträgt. Hilfreich dabei ist die von Frank Krueger erstellte "sqlite-net"-Library.

Sehen wir uns zunächst die wichtigsten Installationsschritte an, bevor wir zur Verwendung der Library kommen:

1. Laden Sie die eigentliche SQLite-DLL herunter. Gegen Sie dazu auf folgende Adresse:

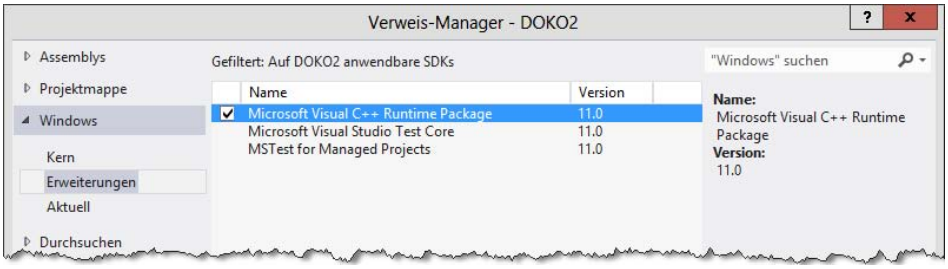
   LINK: `http://www.sqlite.org/download.html`

   In der Rubrik *Precompiled Binaries* ***for Windows Runtime*** wählen Sie entweder die x86- oder die x64-Version (wir verwenden die x86-Version).

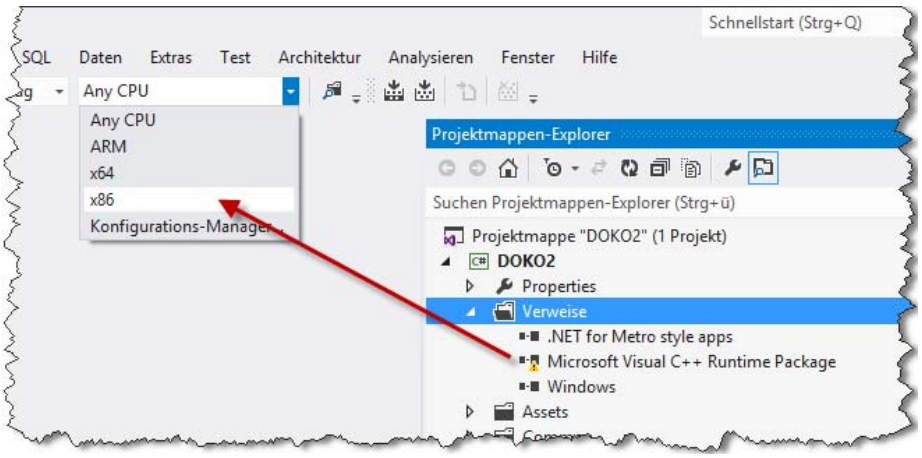
2. Entpacken Sie die heruntergeladene ZIP-Datei.

3. Erstellen oder öffnen Sie jetzt ein App-Projekt und kopieren Sie die *sqlite3.dll* aus der ZIP-Datei in Ihr Projekt. Setzen Sie für die DLL die Eigenschaft *In Ausgabeverzeichnis kopieren* auf *Immer kopieren*.

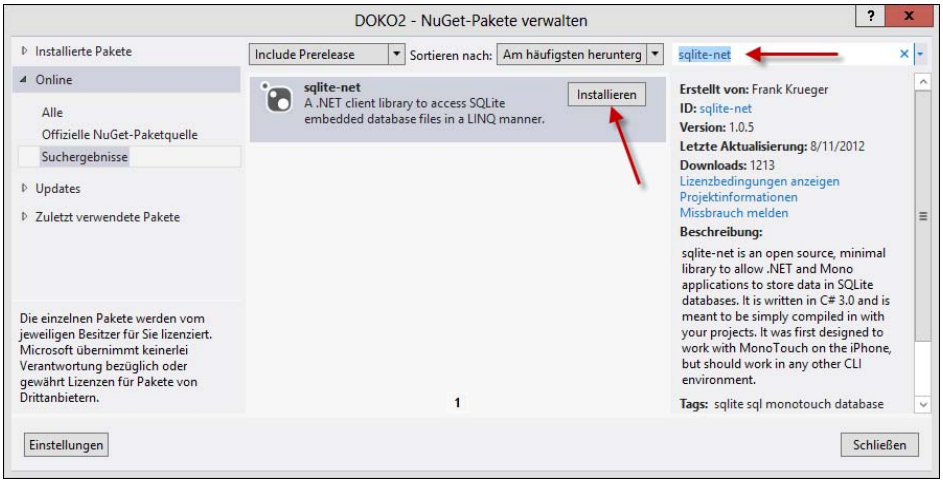
4. Erstellen Sie einen Verweis auf das Microsoft *Visual c++ Runtime Package*:



Nach Einfügen des Verweises werden Sie feststellen, dass dieser mit einem kleinen Warnhinweis versehen ist. Ursache ist, dass wir jetzt nicht mehr plattformunabhängig sind, d h., wir müssen das Projektmappenausgabeziel anpassen (von *Any CPU* auf *x86*):



5. Klicken Sie nun im Menü *Projekte* auf den Punkt *NuGet-Packete* verwalten. Geben Sie in der Suchmaske des folgenden Dialogs "sqlite-net" ein, um das Paket zu suchen. Wählen Sie dann die Schaltfläche *Installieren*:



Nachfolgend wird Ihr App-Projekt um zwei Wrapperdateien (*SQLite.cs* und *SQLiteAsync.cs*) erweitert.

Damit sind alle Vorbereitungen abgeschlossen und wir können uns der eigentlichen Programmierung zuwenden.

21.4.4 Wie kommen wir zu einer neuen Datenbank?

Drei grundsätzliche Varianten bieten sich bei der vorliegenden SQLite-Lösung an:

- Sie erstellen die Datenbank per C#-Code aus vorhandenen Klassen.
- Sie nutzen SQL-Befehle zum Aufbau der Datenbank (oder führen ein Skript aus).
- Sie liefern eine leere Datenbank mit, die Sie beim ersten Start in das App-Datenverzeichnis kopieren (Sie denken bitte daran, dass Sie auf das Installationsverzeichnis nur lesend zugreifen dürfen).

Die folgenden Beispiele zeigen die Grundansätze.

BEISPIEL 21.43: Erstellen der Datenbank aus C#-Klassen zur Laufzeit

C#

```
...
using SQLite;
using Windows.Storage;
using System.Threading.Tasks;
...
    public sealed partial class BasicPage1 : SQL_Einführung.Common.LayoutAwarePage
    {
```

Unsere Verbindung zur Datenbank:

```
        private SQLiteConnection db;
        private TableQuery<Artikel> artikeltabelle;

        public BasicPage1()
        {
            this.InitializeComponent();
            Loaded += BasicPage1_Loaded;
        }
```

Mit dem Laden der Seite wird auch die Datenbank im App-Datenverzeichnis erstellt (ist diese bereits vorhanden, wird sie automatisch geöffnet):

```
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
            db = new SQLite.SQLiteConnection(
                      Path.Combine(ApplicationData.Current.LocalFolder.Path, "firma.db"));
```

Erstellen einer Tabelle:

```
            db.CreateTable<Artikel>();
```

Im Hintergrund wird der Befehl CREATE TABLE IF NOT EXISTS ausgeführt, die Tabelle wird also nicht überschrieben. Die Spalteninformationen und Attribute werden intern anhand des übergebenen Typs bestimmt, an dieser Stelle haben Sie also recht wenig Arbeit.

```
        }
...
```

BEISPIEL 21.43: Erstellen der Datenbank aus C#-Klassen zur Laufzeit

Ganz anders ist das in der Klasse, die später die Datenbank-Entitäten repräsentieren soll. Hier definieren Sie zunächst Eigenschaften wie Sie es gewohnt sind. Nachfolgend steuern Sie über zusätzliche Attribute, welche Eigenschaften die aus den Properties der Klasse generierten Spalten haben sollen:

```
using SQLite;
...
namespace SQL_Einführung
{
```

Unsere Beispiel-Klasse *Artikel*:

```
    public class Artikel : INotifyPropertyChanged
    {
        public Artikel()
        { }
```

Ein Konstruktor zum einfachen Erstellen neuer Instanzen:

```
        public Artikel(string bezeichnung, double preis )
        {
            this._bezeichnung = bezeichnung;
            this._preis = (float) preis;
            this._anzahl = 0;
            this._ausverkauft = true;
        }
```

Unser Primärschlüssel (ein Zählerwert):

```
        private int _id;

        [AutoIncrement, PrimaryKey]
        public int Id
        {
            get { return _id; }
            set {
                _id = value;
                this.NotifyPropertyChanged("Id");
            }
        }
```

Ein indiziertes Textfeld in der Datenbank mit der Länge 100:

```
        private string _bezeichnung;

        [Indexed, MaxLength(100)]
        public string Bezeichnung
        {
            get { return _bezeichnung; }
            set {
                _bezeichnung = value;
```

BEISPIEL 21.43: Erstellen der Datenbank aus C#-Klassen zur Laufzeit

```
            this.NotifyPropertyChanged("Bezeichnung");
        }
    }
```

Ein Float-Feld:

```
    private float _preis;
    public float Preis
    {
        get { return _preis; }
        set {
            _preis = value;
            this.NotifyPropertyChanged("Preis");
        }
    }
```

Ein Integer-Feld:

```
    private int _anzahl;
    public int Anzahl
    {
        get { return _anzahl; }
        set {
            _anzahl = value;
            this.NotifyPropertyChanged("Anzahl");
        }
    }
```

Ein Boolean-Feld:

```
    private bool _ausverkauft;
    public bool Ausverkauft
    {
        get { return _ausverkauft; }
        set {
            _ausverkauft = value;
            this.NotifyPropertyChanged("Ausverkauft");
        }
    }
```

Dieses Feld wird nur in der App benötigt, es gibt keine Entsprechung in der Datenbank:

```
    [Ignore]
    public string ID_Bezeichnung
    {
        get
        {
            return _id.ToString() + "_" + Bezeichnung;
        }
    }
```

BEISPIEL 21.43: Erstellen der Datenbank aus C#-Klassen zur Laufzeit

Für die Anzeige im Listenfeld eine überschriebene *ToString*-Methode:

```
        public override string ToString()
        {
            return string.Format("Id:{0} Bez:{1} Preis:{2}", _id, _bezeichnung, _preis);
        }
...
```

Wie Sie sehen, besteht Ihre Hauptarbeit im Definieren der Mapperklassen, folgende Attribute sind verfügbar:

- *PrimaryKey*
  Kennzeichnet das Feld als Primärschlüssel.
- *AutoIncrement*
  Erstellt ein Zählerfeld (meist gleichzeitig der Primärschlüssel).
- *Indexed*
  Das Feld soll indiziert werden.
- *MaxLength(<anzahl>)*
  Das Textfeld soll eine maximale Länge von <anzahl> haben. Der Standardwert sind 140 Zeichen.
- *Ignore*
  Dieses Feld wird nicht in der Datenbank gespeichert

Folgende Datentypen werden durch den Mapper unterstützt:

- Integer
- Boolean (intern Integer mit 1=*true*)
- Enumerations (intern *Integer*)
- Gleitkommawerte (intern als *float*)
- String (intern *varchars* mit durch *MaxLength* festgelegter Länge)
- *DateTime*

Ja, ja, einen Enterprise-SQL-Server können Sie so nicht ersetzen, aber das dürfte in den meisten Fällen auch gar nicht Ihr Ziel sein.

Neben obigem Weg gibt es auch einen anderen: Sie erstellen die Datenbank bereits fix und fertig und liefern eine leere Version davon mit der App aus. Diese Datenbank befindet sich zunächst im Installationsverzeichnis der App, das bekanntermaßen schreibgeschützt ist. Sie müssen also nur Sorge dafür tragen, diese Datei in das App-Datenverzeichnis zu kopieren.

BEISPIEL 21.44: Datenbank-Vorlage aus dem Installations- in das Datenverzeichnis kopieren

C#

```
using SQLite;
using System.Diagnostics;
using Windows.Storage;
using Windows.ApplicationModel;
...
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
            if (!await ApplicationData.Current.LocalFolder.FileExistsAsync("firma.db"))
            {
                StorageFile file = await
                   Package.Current.InstalledLocation.GetFileAsync("Vorlage_firma.db");
                await file.CopyAsync(ApplicationData.Current.LocalFolder, "firma.db");
            }
            db = new   SQLite.SQLiteConnection(Path.Combine(
                       ApplicationData.Current.LocalFolder.Path, "firma.db"));
...
```

Vermutlich werden Sie vergebens nach der Methode *FileExistsAsync* Ausschau halten, denn diese schreiben wir uns selbst:

```
static class Erweiterungsmethoden
  {
      public async static Task<bool> FileExistsAsync(this StorageFolder folder,
                                                     string filename)
      {
          try
          {
              await folder.GetFileAsync(filename);
              return true;
          }
          catch (Exception)
          { return false; }
      }
  }
```

HINWEIS: Mit dem SQLite-Administrator (siehe *http://sqliteadmin.orbmu2k.de/*) können Sie die Datenbanken komfortabel entwerfen.

21.4.5 Wie werden die Daten manipuliert?

An dieser Stelle halten wir uns mit weitschweifigen Ausführungen zurück, ein kleines Beispiel soll genügen.

HINWEIS: Wesentlich mehr zum Thema Datenbanken finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012] aus dem Microsoft Press-Verlag.

BEISPIEL 21.45: Manipulieren der SQLite-Datenbank

C#

```
...
        async void BasicPage1_Loaded(object sender, RoutedEventArgs e)
        {
...
            db = new SQLite.SQLiteConnection(
                  Path.Combine(ApplicationData.Current.LocalFolder.Path, "firma.db"));
            db.CreateTable<Artikel>();  // ein CREATE TABLE IF NOT EXISTS
```

Wir fragen die Anzahl der Datensätze in der Tabelle *Artikel* ab:

```
            int anzahl = db.ExecuteScalar<int>("SELECT COUNT(*) FROM artikel");
```

Sind keine Datensätze vorhanden, erstellen wir vier neue Einträge:

```
            if (anzahl == 0)
            {
                db.RunInTransaction(() =>
                    {
                        db.Insert(new Artikel("Kuchenhörnchen", 1.45));
                        db.Insert(new Artikel("Brötchen", 0.46));
                        db.Insert(new Artikel("Brot", 1.99));
                        db.Insert(new Artikel("Eis rot", 1.35));
                    });
            }
```

Wir fragen die komplette Tabelle ab und zeigen diese an:

```
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
...
```

Einen einzelnen Artikel einfügen und die Tabelle erneut abfragen:

```
        private void Button_Click_2(object sender, RoutedEventArgs e)
        {
```

Artikel erzeugen:

```
            Artikel art = new Artikel("Weissbrot", 1.66);
```

In der Datenbank speichern:

```
            db.Insert(art);
```

Den Autowert abfragen:

```
            Debug.WriteLine("Der neue Autowert ist: " + art.Id.ToString());
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
```

BEISPIEL 21.45: Manipulieren der SQLite-Datenbank

Wir löschen den ersten Artikel in der Liste:

```
        private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            db.Delete(artikeltabelle.First());
```

Tabelle erneut abfragen und anzeigen:

```
            artikeltabelle = db.Table<Artikel>();
            ListBox1.ItemsSource = artikeltabelle;
        }
```

Wir fragen alle Artikel ab, die einen Preis größer 1,4 Euro haben:

```
        private void Button_Click_4(object sender, RoutedEventArgs e)
        {
           ListBox1.ItemsSource = db.Query<Artikel>(
                                              "SELECT * FROM artikel WHERE preis > 1.4");
        }
...
```

21.5 Vertrieb der App

Kommen wir noch kurz zum Thema "Distribution von Windows 8-Apps". An dieser Stelle werden wir es unterlassen, uns in epischer Länge mit all den Restriktionen, Vorgaben, Preismodellen und Lizenzierungsarten für die Distribution von Apps über den Windows Store zu befassen. Auch den Prozess nach der Übertragung des App-Package an den Store lassen wir "links liegen", hier sind in der nächsten Zeit sicher noch Änderungen zu erwarten.

Wir wollen hingegen kurz auf die beiden relevanten Varianten der App-Distribution eingehen:

- Vertrieb über den Windows Store (kostenlos oder kostenpflichtig),
- Distribution/Installation in firmeneigenen Netzwerken per Sideloading.

21.5.1 Verpacken der App

Erster Schritt beim Verteilen ist das Verpacken der App in einem Container, dem so genannten App-Package. Bei diesem handelt es sich eigentlich um eine ZIP-Datei, die aus der EXE, den Ressourcen, der Signatur, sowie der Manifestdatei besteht.

HINWEIS: Grundvoraussetzung für das weitere Vorgehen ist zunächst eine fehlerfrei[1] lauffähige App, die als **Release** kompiliert wurde.

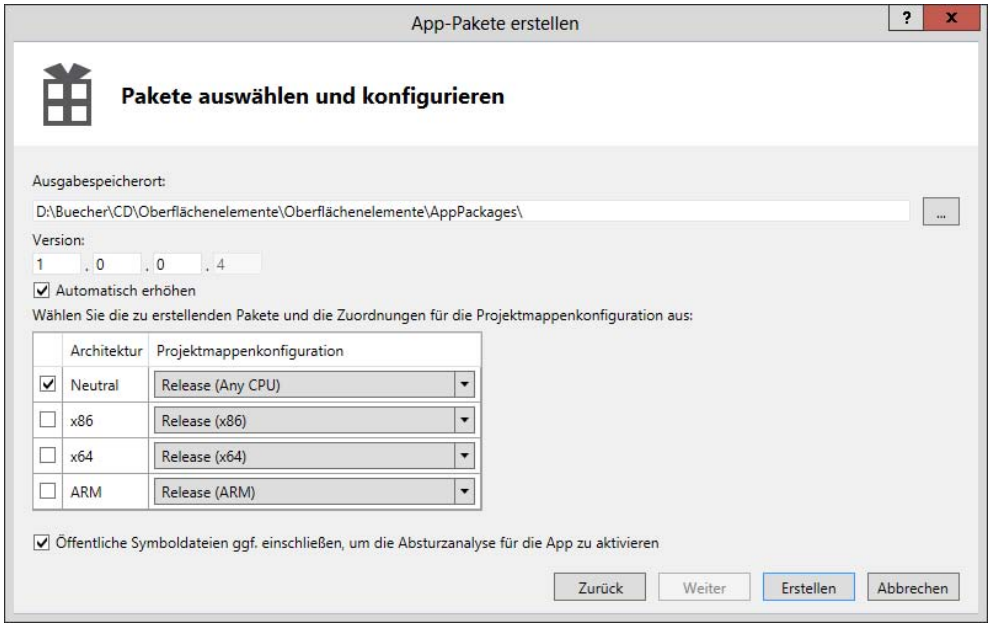
[1] So weit man dies bei Software überhaupt sagen kann.

Sie könnten natürlich alle Schritte zum fertigen Package auch an der Kommandozeile realisieren (VB-/C#C#-Compiler, *MakeAppx.exe*, *Signtool.exe*), wir gehen jedoch ganz einfach mal davon aus, dass Sie mit Visual Studio arbeiten und sich so eine Menge Arbeit ersparen.

Erstellen Sie also zunächst am besten die Projektmappe neu (*Erstellen|Projektmappe neu erstellen*) und achten Sie darauf, dass die Konfiguration auf *Release* gestellt ist.

Wählen Sie nun den Menüpunkt *Projekt|Store|App-Pakete erstellen*. Es erscheint ein entsprechender Assistent, der Sie durch die einzelnen Etappen führt:

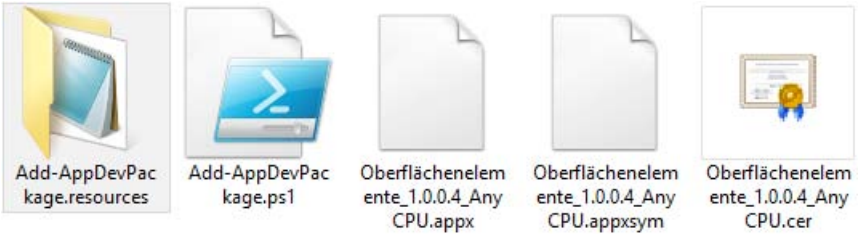
Hier haben Sie die Auswahl, ob Sie einen Paket für den Shop oder für eine lokale Distribution erstellen wollen. Wählen Sie zunächst die zweite Variante. Nachfolgend wählen Sie das Ausgabeverzeichnis, die Versionsnummer und die Architektur aus:



Nach dem Klick auf "Erstellen" ist nach wenigen Sekunden das Verpacken abgeschlossen, und Sie können die Dateien im Ausgabeverzeichnis in Augenschein nehmen.

HINWEIS: Alternativ können Sie jetzt auch eine Prüfung durch das *Windows App Certification Kit* starten, siehe dazu den folgenden Abschnitt 21.5.2.

Im Ausgabeverzeichnis finden Sie eine Uploadpackage und ein Unterverzeichnis vor, in dem alle Dateien für eine lokale Distribution enthalten sind:



Die Dateien im Einzelnen:

- ein Powershell-Skript zur Installation (*Add-AppDevPackage.ps1*)
- das eigentliche App-Package (**.appx*)
- die optionale Symboldatei für die App (**.appsym*)
- ein Zertifikat für die spätere Installation der App (**.cer*)

Bevor wir zur Installation mittels des obigen PowerShell-Skripts kommen, wollen wir uns noch die Verwendung des *Windows App Certification Kit* ansehen.

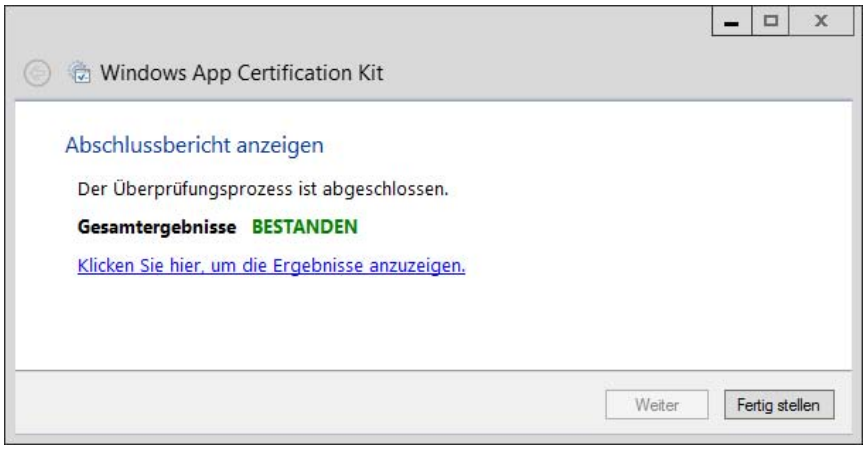
21.5.2 Windows App Certification Kit

Um eine erste Prüfung der App bereits auf dem PC des Entwicklers zu ermöglichen, stellt Microsoft den Entwicklern das *Windows App Certification Kit* zur Verfügung. So müssen Sie für eine erste Prüfung auf Kompatibilität die App nicht erst im Shop einreichen, sondern können diesen Prozess schon einmal vorab auf Ihrem System durchlaufen. Rufen Sie die Anwendung entweder direkt auf oder aus dem Package-Assistenten heraus.

Wählen Sie nach dem Start den ersten Menüpunkt "Windows Store-App prüfen". Nachfolgend durchsucht das Certification Kit Ihr System und listet alle verfügbaren Apps auf. Wählen Sie Ihre App aus und starten die den Test-Prozess.

HINWEIS: Nachfolgend sollten Sie am besten eine Tasse Kaffee trinken und den PC nicht berühren, Ihre App wird mehrfach gestartet und beendet, Tasteneingaben in anderen Anwendungen könnten das Testergebnis verfälschen.

Nach einige Zeit werden Sie zur Eingabe eines Dateinamens aufgefordert, in dieser Datei werden die Ergebnisse der Prüfung gesichert. Das endgültige Ergebnis wird Ihnen bereist hier angezeigt:



Das Testergebnis im Einzelnen (Ausschnitt):

Windows App Certification Kit - Test Results

App-Name:	1fc01a7c-1433-4074-ab86-517c59b8d321
App-Version:	1.0.0.5
App-Herausgeber:	Thomas
Betriebssystemversion:	Microsoft Windows 8 Pro (6.2.9200.0)
Zeitpunkt der Berichtgenerierung:	29.08.2012 11:49:34

Gesamtergebnis: BESTANDEN

Test von Abstürzen oder Hängen

BESTANDEN	App-Starttests
BESTANDEN	Abstürze und Hängen

Test der Einhaltung des App-Manifests

BESTANDEN	App-Manifest

Test der Windows-Sicherheitsfunktionen

BESTANDEN	Analyzer für Binärdateien

Test der unterstützten APIs

BESTANDEN	Unterstützte APIs

Nutzen Sie die Hinweise des Tests, um Ihre App für den erfolgreichen Vertrieb vorzubereiten. Verwenden Sie möglichst einen Computer (bzw. Tablet), der am unteren Ende der Leistungsskala steht (Bildschirmgröße, Arbeitsspeicher, Prozessor). Nur so können Sie sicherstellen, dass Ladezeiten eingehalten werden und Darstellungsgrößen ausreichend sind. Testen Sie in jedem Fall auch die reine Toucheingabe bei unterschiedlichen Bildschirmgrößen, um eine exakte und einfache Bedienung zu garantieren.

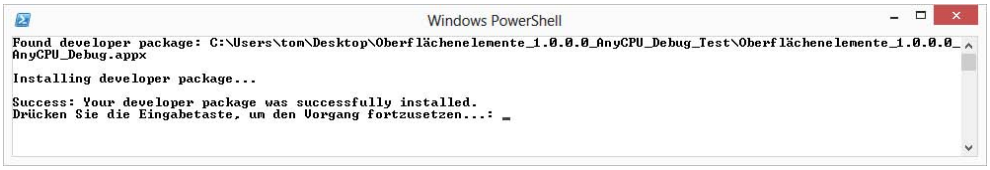
HINWEIS: Wollen Sie Apps professionell entwickeln und weitergeben, müssen Sie sich ein eigenes Zertifikat von einer vertrauenswürdigen Zertifizierungsstelle besorgen. Dieses können Sie dann über den Manifestdatei-Assistenten (Seite *Verpacken*) aus dem Zertifikatsspeicher dem Projekt zuweisen. Andernfalls arbeiten Ihre Projekte nur mit einem temporären Zertifikat, das Sie nicht für eine Weitergabe nutzen sollten.

21.5.3 App-Installation per Skript

Geht es um die eigentliche Weitergabe, können Sie Ihre App unter Verwendung eines Entwicklerkontos im Windows Store einreichen. Ein alternativer Weg steht Ihnen als Entwickler und Firmenkunde offen. Die Rede ist von der sogenannten Sideloading-Installation, die ohne den Windows Store auskommt. Bevor Sie sich jetzt zu früh freuen, sollten Sie einen Blick auf die Voraussetzungen werfen:

- Die App muss signiert sein.
- Der Ziel-PC muss dem Zertifikat vertrauen.
- Die Gruppenrichtlinie "Allow all trusted applications to install" muss aktiviert sein.
- Der Ziel-PC muss entweder über eine Entwicklerlizenz verfügen (diese kann jederzeit abgerufen werden, ist aber nicht lange gültig) oder Sie verfügen über einen entsprechend freigeschalteten PC (Windows 8 Enterprise Edition und Windows 8 Server Edition), der einer Domain zugeordnet ist. Andere Windows 8 Versionen müssen mit einem speziellen Key für die Installation freigeschaltet werden.

Sind diese Voraussetzungen erfüllt, können Sie das vom Package-Assistenten erstellte Verzeichnis auf den Ziel-PC kopieren und durch den Aufruf des PowerShell-Skripts installieren. Nach etlichen Sicherheitsabfragen und Bestätigungen (Änderung Gruppenrichtlinie, Installation des Zertifikats) ist es dann so weit, Ihre App wurde installiert:



Alternativ können Sie auch App-Packages in Windows-Installationen integrieren, aber dies würde hier zu weit führen.

21.6 Ein Blick auf die App-Schwachstellen

So schön die Entwicklung und der Vertrieb von Windows 8-Apps auch sein mag, so unangenehm sind leider auch einige Schwachstellen, die sich aus Sicht des Entwicklers offenbaren. Zwar sind die Apps untereinander und auch vom System vorbildlich getrennt, die App selbst ist aber relativ schutzlos auf dem Zielsystem des Anwenders installiert. Dies betrifft zwei Schwerpunkte:

- das Installationsverzeichnis
- das App-Datenverzeichnis

Sehen wir uns diese einmal genauer an.

21.6.1 Quellcodes im Installationsverzeichnis

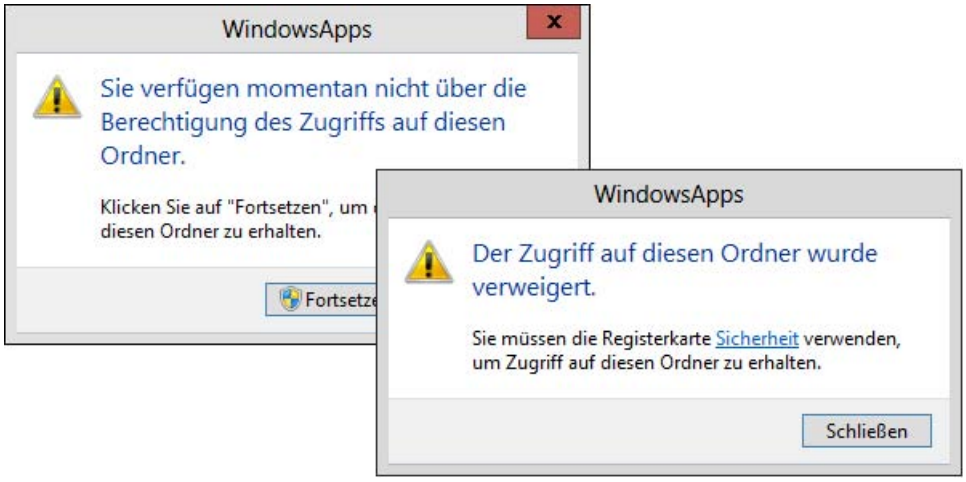
Haben Sie schon .NET-Apps entwickelt, ist Ihnen eine Schwachstelle von .NET-Programmen sicher geläufig: Mit geeigneten Tools (.NET-Reflector, ILSpy etc.) können Sie problemlos einen Blick in die Arbeit anderer Programmierer werfen. Sie haben die freie Wahl der Programmiersprache, lediglich die Variablennamen bleiben bei dieser Form der Neugier auf der Strecke. Leider ist dies für aufwändige Algorithmen, die in mühseliger Kleinarbeit erstellt wurden, nicht unbedingt immer gewünscht. Wie sieht es eigentlich mit dieser Problematik im Zusammenhang mit den Windows 8-Apps aus?

Ja, also ... im Prinzip ... ähm ... also um es kurz zu machen, versuchen Sie mal auf das Verzeichnis

```
C:\Program Files\WindowsApps\
```

zuzugreifen.

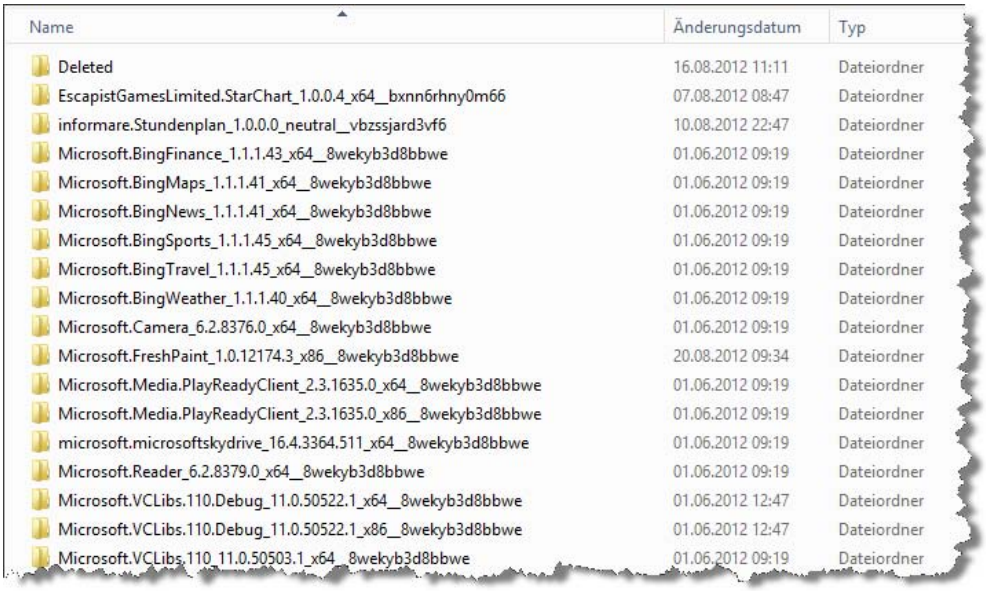
Folgende Meldung begrüßt Sie:



Das sieht zunächst recht gut aus, doch lassen Sie sich von dem Geplänkel nicht abschrecken:

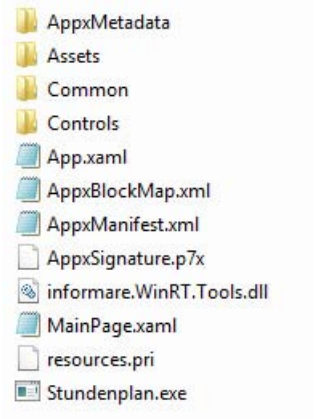
- Setzen Sie die Gruppe *Administratoren* als Besitzer des Ordners ein[1].
- Übernehmen Sie alle Rechte für die Gruppe *Administratoren* (Vollzugriff).
- Greifen Sie auf das Verzeichnis mit vollen Rechten zu.

Ein Blick hinein:



Name	Änderungsdatum	Typ
Deleted	16.08.2012 11:11	Dateiordner
EscapistGamesLimited.StarChart_1.0.0.4_x64__bxnn6rhny0m66	07.08.2012 08:47	Dateiordner
informare.Stundenplan_1.0.0.0_neutral__vbzssjard3vf6	10.08.2012 22:47	Dateiordner
Microsoft.BingFinance_1.1.1.43_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.BingMaps_1.1.1.41_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.BingNews_1.1.1.41_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.BingSports_1.1.1.45_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.BingTravel_1.1.1.45_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.BingWeather_1.1.1.40_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.Camera_6.2.8376.0_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.FreshPaint_1.0.12174.3_x86__8wekyb3d8bbwe	20.08.2012 09:34	Dateiordner
Microsoft.Media.PlayReadyClient_2.3.1635.0_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.Media.PlayReadyClient_2.3.1635.0_x86__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
microsoft.microsoftskydrive_16.4.3364.511_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.Reader_6.2.8379.0_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner
Microsoft.VCLibs.110.Debug_11.0.50522.1_x64__8wekyb3d8bbwe	01.06.2012 12:47	Dateiordner
Microsoft.VCLibs.110.Debug_11.0.50522.1_x86__8wekyb3d8bbwe	01.06.2012 12:47	Dateiordner
Microsoft.VCLibs.110_11.0.50503.1_x64__8wekyb3d8bbwe	01.06.2012 09:19	Dateiordner

Ah, oh – das sind ja alle installierten Anwendungen! Aber es kommt noch besser. Im folgenden Beispiel haben wir uns mit der .NET-App *Stundenplan* näher beschäftigt:



[1] Wer nicht weiß, wie es geht, sollte die Finger davon lassen ...

- Die im Verzeichnis enthaltene EXE und die DLL können mit ILSpy **komplett** und problemlos analysiert werden.
- Alle *Pages*, *UserControls* etc. liegen als **einzelne XAML-Dateien** in diesem Verzeichnis und können mit einem normalen Texteditor durchgesehen bzw. bearbeitet werden.
- Alle Ressourcen (Bilder, Styles) sind als einzelne Dateien frei zugänglich.

Mit obigen Informationen dürfte es keine Stunde dauern, um ein kompilierfähiges Projekt auf dem eigenen PC zu erstellen. Da freut sich der Plagiator, schade um das darin steckende Know-how ...!

Alternativ können Sie ja auch mal an den XAML-Dateien herumbasteln und diese speichern. Nach einem Neustart der App werden Sie feststellen, dass Ihre Änderungen übernommen wurden. Derartige "Nachbearbeitungen" dürften wohl kaum im Sinne des Entwicklers sein.

21.6.2 Zugriff auf den App-Datenordner

Neben obiger Schwachstelle sollten Sie sich auch mit dem Fakt vertraut machen, dass Ihre App zwar nicht überall zugreifen darf, dies gilt jedoch nicht für den Anwender. Dieser kann ganz schnell auf den App-Datenordner (*Users*\<*Anwendername*>*AppData**Local**Packages*\<*App-ID*>) zugreifen.

Ist es für den Entwickler naheliegend, in diesem Ordner zum Beispiel Initialisierungsdaten, Nutzerdaten etc. abzulegen, kann dies durch einen Nutzereingriff schnell zum Problem für den Entwickler werden.

Stellen Sie sich einfach das Szenario eines Spiele-Entwicklers vor, der für eine Test-App mit begrenzter Levelzahl/Punktezahl Spielstände in diesem Ordner ablegt. Allein durch das Wiederherstellen, Löschen oder Verändern der Daten könnte der Anwender z.B. unbegrenzt mit dieser App spielen. Da hilft nur das Verschlüsseln der Daten, um unerwünschte Nutzung zu verhindern.

21.7 Praxisbeispiele

21.7.1 Ein Verzeichnis auf Änderungen überwachen

Wer kennt das nicht aus alten Win32-Zeiten – die Möglichkeit, Änderungen von Dateien in einem Verzeichnis zu überwachen. Einfachster Fall: Sie zeigen eine Liste aller Bilder in einem Verzeichnis an. Zwischenzeitlich schaltet der Nutzer zu einer anderen App um (oder zum Desktop) und eine der Dateien wird gelöscht bzw. es wird eine neu hinzugefügt. Ihre Liste ist jetzt nicht mehr aktuell. Genau für diesen Fall bietet sich eine ereignisgesteuerte Verzeichnisüberwachung an, die wir Ihnen im Weiteren vorstellen möchten.

Oberfläche

Fügen Sie einer Seite lediglich eine *ListBox* und zwei Schaltflächen hinzu.:

```
            <StackPanel>
                <StackPanel Orientation="Horizontal">
                    <Button Click="Button_Click_2">Neuen Ordner erzeugen</Button>
                    <Button Click="Button_Click_3">Neue Datei erzeugen</Button>
                </StackPanel>
                <ListBox Name="ListBox1"/>
            </StackPanel>
```

Quellcode

Öffnen Sie die Datei *BasicPage1.xaml.cs* und nehmen Sie folgende Erweiterungen vor:

```
...
using Windows.Storage;
using System.Diagnostics;
using Windows.Storage.Search;
...
    public sealed partial class BasicPage1 : Verzeichnisüberwachen.Common.LayoutAwarePage
    {
```

Eine Instanz unserer *StorageFileQuery*:

```
        StorageFileQueryResult query;
```

Mit dem Öffnend der Seite erzeugen wir zunächst eine Dateilistenabfrage für das App-Datenverzeichnis:

```
        async protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            QueryOptions queryOptions = new QueryOptions(CommonFileQuery.DefaultQuery, null);
```

Mit Unterverzeichnissen:

```
            queryOptions.FolderDepth = FolderDepth.Deep;
            queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable;
```

Abfrage erzeugen:

```
            query = ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(
                                                                        queryOptions);
```

Ereignishandler registrieren:

```
            query.ContentsChanged += query_ContentsChanged;
```

Dieser Aufruf ist wichtig, die Überwachung wird erst nach diesem Aufruf gestartet:

```
            await query.GetFilesAsync();
```

Hier finden Sie den Dateipfad für manuelle Änderungen per Explorer:

```
            Debug.WriteLine("Ändern Sie das Verzeichnis: " +
                            ApplicationData.Current.LocalFolder.Path);
        }
```

Unser Ereignishandler (wir müssen auf den Vordergrundthread umschalten, das Ereignis wird extern ausgelöst):

```
        async void query_ContentsChanged(IStorageQueryResultBase sender, object args)
        {
            await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, async () =>
            {
```

Wir rufen die neue Dateiliste ab:

```
                ListBox1.Items.Clear();
                foreach (var f in await query.GetFilesAsync())
                    ListBox1.Items.Add("Änderung: " + f.Path);
            });
        }
```

Ein Verzeichnis testweise erzeugen:

```
        async private void Button_Click_2(object sender, RoutedEventArgs e)
        {
            StorageFolder folder = await
                  ApplicationData.Current.LocalFolder.CreateFolderAsync("Testordner",
                  CreationCollisionOption.GenerateUniqueName);
        }
```

Eine Datei testweise erzeugen:

```
        async  private void Button_Click_3(object sender, RoutedEventArgs e)
        {
            StorageFile file = await
                   ApplicationData.Current.LocalFolder.CreateFileAsync("Hallo.txt",
                   CreationCollisionOption.GenerateUniqueName);

        }
```

Test

Starten Sie die App und klicken Sie auf die beiden Schaltflächen. Mit etwas Verzögerung sollte die Änderung durch Ihre App registriert werden, die Dateiliste wird aktualisiert.

Sie können die Änderungen auch vom Desktop aus vornehmen, den Verzeichnispfad finden Sie im Debug-Fenster. Doch hier lauert dann auch schon das Problem:

HINWEIS: Läuft Ihre App im Hintergrund, werden Änderungen erst registriert, wenn die App wieder im Vordergrund steht (Sie erinnern sich ... Suspended). Stehen Ihnen zwei Bildschirme zur Verfügung, können Sie Ihre App im einen und den Desktop im anderen Bildschirm darstellen. Änderungen per Desktop wirken sich jetzt auch sofort auf die App aus (diese ist ja im Vordergrund).

21.7.2 Eine App als Freigabeziel verwenden

Soll Ihre App Daten von anderen Apps empfangen, können Sie entweder die Zwischenablage einsetzen oder Sie nutzen die *Teilen*-Funktion. Während das eigentliche Teilen, d.h. Freigeben der Daten, in weiten Teilen mit der Verwendung der Zwischenablage übereinstimmt, müssen Sie beim Zugriff auf die geteilten Daten schon etwas mehr Aufwand treiben.

Unser Beispielprogramm beschränkt sich auf die Anzeige von

- Dateilisten,
- Texten
- und Bitmap-Daten.

Oberfläche

Erstellen Sie eine neue App und fügen Sie zwei Seiten hinzu. *BasicPage1.xaml* stellt die standardmäßig angezeigte Seite dar, wie sie beim normalen Starten der App angezeigt wird. *InsertPage.xaml* ist die Bedienoberfläche, die im Teilen-Flyout erscheint. Hier zeigen wir die übergebenen Daten mittels *ListBox* und *Image*-Control an:

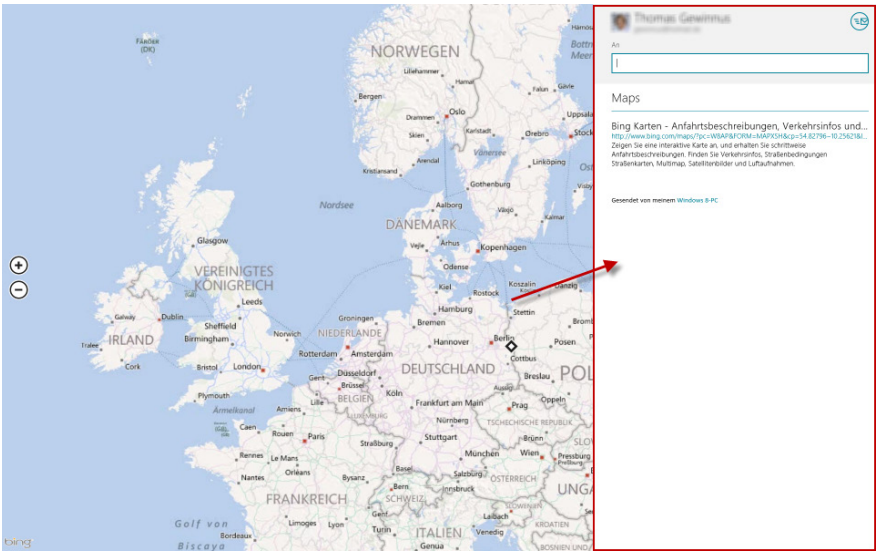
```
<ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
              Margin="50" Grid.Row="0" ZoomMode="Disabled">
          <StackPanel>
              <ListView Name="ListBox1" Width="400" Height="200"/>
              <Image Name="Image1" Height="200" />
              <Button Click="Button_Click_2" Name="Button2">Verarbeiten</Button>
          </StackPanel>

      </ScrollViewer>
```

Mit der Schaltfläche wird später die Share-Operation abgeschlossen.

Legen Sie das Layout der Seite so an, dass Sie mit dem schmaleren Flyout-Fenster klarkommen, in welchem Ihre App angezeigt wird, hier haben Sie ausnahmsweise mal nicht den kompletten Bildschirm für sich zur Verfügung (Beispiel siehe folgende Abbildung).

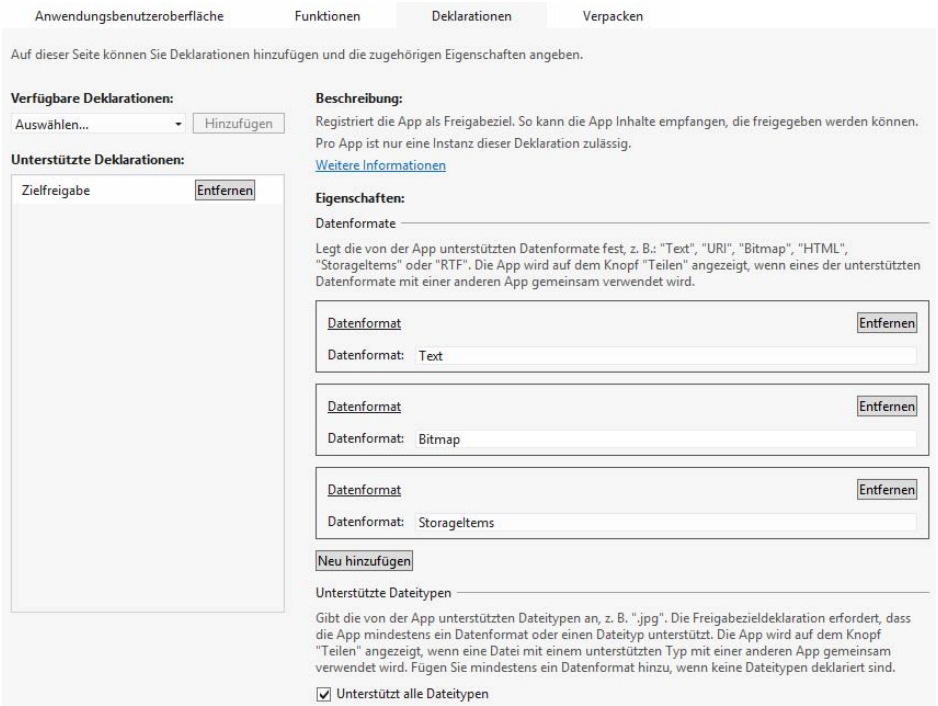
Nachfolgend kommt der größere Teil der Arbeit auf Sie zu, das Auswerten der Daten.

HINWEIS: Hier sollten Sie mit Fehlerbehandlung nicht sparen, ob die gelieferten Daten korrekt sind, stellt sich nämlich erst hier heraus.



Registrieren der App als Freigabeziel

Um Ihre App als Datenempfänger zu kennzeichnen, muss der Manifestdatei *Package.appxmanifest* zunächst die Deklaration *Zielfreigabe* hinzugefügt werden. Wählen Sie in der linken Liste den entsprechenden Eintrag und klicken Sie auf "Hinzufügen":



In obiger Eingabemaske können Sie jetzt gleich noch die von Ihrer App unterstützten Datenformate hinzufügen. Ihre App wird später nur dann im Charmbar angezeigt, wenn die freigegebenen Daten mit den obigen Datentypen übereinstimmen.

Sollen auch Dateien (Datenformat *StorageItems*) ausgetauscht werden, sollten Sie zusätzlich definieren, welche Dateitypen bei dieser Operation zulässig sind. Hier können Sie entweder die Typen einzeln aufführen oder Sie setzen ein Häkchen bei "Unterstützt alle Dateitypen".

Damit ist Ihre App aus WinRT-Sicht als Ziel von Datenfreigaben vorbereitet, Ihre App würde jetzt auch schon als Ziel angezeigt werden. Doch es fehlt noch der eigentliche Eintrittspunkt in die App, d.h. die Methode, die im Zusammenhang mit der Share-Operation in Ihrer App aufgerufen wird.

Quellcode (App.xaml.cs)

Wir implementieren zunächst den Eintrittspunkt *OnShareTargetActivated*. Öffnen Sie die Datei *App.xaml.cs* und fügen Sie folgende Methode hinzu:

```
        protected override void OnShareTargetActivated(ShareTargetActivatedEventArgs args)
        {
            var rootFrame = new Frame();
            rootFrame.Navigate(typeof(InsertPage), args.ShareOperation);
            Window.Current.Content = rootFrame;
            Window.Current.Activate();
        }
```

Unabhängig von der aktuell angezeigten Seite öffnen wir eine neue Seite für die Anzeige der Share-Informationen. Als Parameter wird ein *ShareOperation*-Objekt übergeben, über dieses werten wir dann die übergebenen Daten aus.

Quellcode (InsertPage.xaml.cs)

In unserer Beispiel-App wollen wir die Daten lediglich in einer *ListBox* bzw. einem *Image*-Control anzeigen, um das Grundprinzip der Datenabfrage zu demonstrieren.

Die obligatorischen Namespaces rund um den Datentransfer:

```
...
using Windows.ApplicationModel.DataTransfer;
using Windows.ApplicationModel.DataTransfer.ShareTarget;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Media.Imaging;
...
    public sealed partial class InsertPage : Zielfreigabe_Bsp.Common.LayoutAwarePage
    {
```

Das folgende Objekt nimmt das an die Seite übergebene *ShareOperation*-Objekt auf:

```
        private ShareOperation shareop;
```

Mit dem Öffnen der Seite rufen wir zunächst den Übergabeparameter ab und speichern diesen:

```
        async protected override void OnNavigatedTo(NavigationEventArgs e)
```

```
        {
            shareop = (ShareOperation)e.Parameter;
```

Dem System mitteilen, dass wir die Share-Operation gestartet haben:

```
            shareop.ReportStarted();
```

Wir werten die Informationen über den Datenlieferanten aus:

```
            ListBox1.Items.Add("Title: " + shareop.Data.Properties.Title);
            ListBox1.Items.Add("Description: " + shareop.Data.Properties.Description);
```

Auswerten auf Textformat:

```
            if (shareop.Data.Contains(StandardDataFormats.Text))
               {
                   try
                   {
```

Text anzeigen:

```
                      ListBox1.Items.Add("Text = " + await shareop.Data.GetTextAsync());
                   }
                   catch (Exception ex)
                   {
                       ListBox1.Items.Add("Error: " + ex.Message);
                   }
               }
```

Auswerten, ob eine Dateiliste übergeben wurde:

```
            if (shareop.Data.Contains(StandardDataFormats.StorageItems))
            {
                ListBox1.Items.Add("Dateien: ");
                try
                {
```

Die Dateiliste durchlaufen und die kompletten Dateinamen anzeigen:

```
                    var items = await shareop.Data.GetStorageItemsAsync();
                    foreach (var item in items)
                        ListBox1.Items.Add(item.Path);
                }
                catch (Exception ex)
                {
                    ListBox1.Items.Add("Error: " + ex.Message);
                }
            }
```

Wie schon bei der Zwischenablage macht uns auch hier die Bitmap die meiste Arbeit:

```
            if (shareop.Data.Contains(StandardDataFormats.Bitmap))
            {
                ListBox1.Items.Add("Bitmap: ");
```

```
            try
            {
```

IRandomAccessStreamReference abrufen, *BitmapImage* erstellen, Stream als Datenquelle für das *BitmapImage* erstellen und zuweisen, *BitmapImage* als Datenquelle für das *Image*-Control festlegen:

```
                IRandomAccessStreamReference imgdata = await shareop.Data.GetBitmapAsync();
                if (imgdata != null)
                {
                    BitmapImage bi = new BitmapImage();
                    bi.SetSource(await imgdata.OpenReadAsync());
                    Image1.Source = bi;
                }
            }
            catch (Exception ex)
            {
                ListBox1.Items.Add("Error: " + ex.Message);
            }
        }
```

Hier können wir dem System die erfolgreiche Datenübernahme mitteilen:

```
        shareop.ReportDataRetrieved();
    }
```

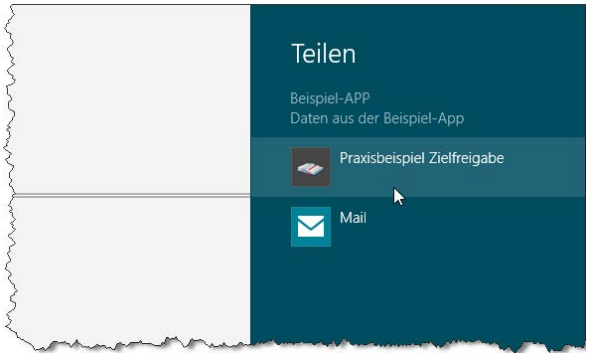
Im Normalfall wird der Nutzer eine Aktion in diesem Fenster auslösen (Email versenden etc.), wir belassen es beim Ändern der Schaltflächenbeschriftung. Mit dem Aufruf *ReportCompleted* wird die Seite geschlossen, die Share-Operation ist abgeschlossen:

```
    private void Button_Click_2(object sender, RoutedEventArgs e)
    {
        Button2.Content = "Verarbeitet ...";
        shareop.ReportCompleted();
    }
}
```

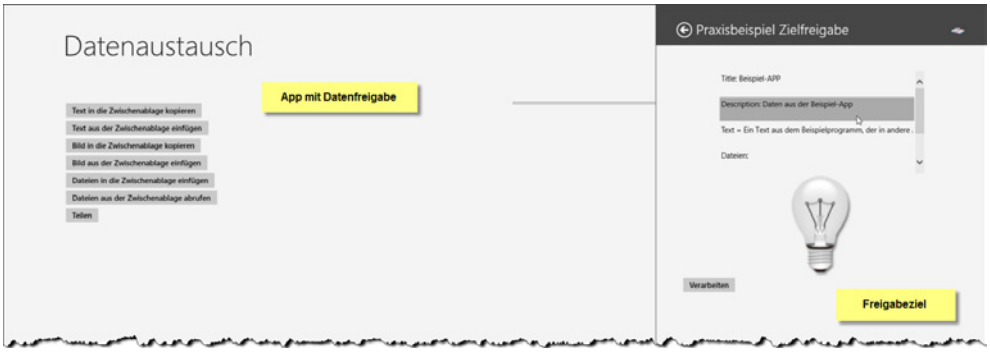
Test

Starten Sie die App zunächst einmal, um sie im System zu registrieren. Sie können sie nachfolgend entweder schließen oder geöffnet lassen. Starten Sie jetzt eine App mit Unterstützung für die Teilen-Funktion (z.B. unser Beispielprogramm aus dem Abschnitt 21.2.2).

Rufen Sie den Charmbar und die Funktion *Teilen* auf, sollte unsere App angezeigt werden:



Klicken Sie darauf, wird nach einem Augenblick unsere App in einem modalen Flyout präsentiert:



Die Daten sollten entsprechend ausgewertet und angezeigt werden.

21.7.3 ToastNotifications einfach erzeugen

Suchen Sie einen einfachen und vor allem komfortablen Weg um Popup-Benachrichtigungen anzuzeigen, so schreiben Sie sich doch einfach eine eigene Klasse dafür.

HINWEIS: Die Grundlagen und die Verwendung dieser Klasse werden im Abschnitt 21.3.2 ab Seite 1089 beschrieben.

Unsere Aufgabe: Abhängig von den übergebenen Eigenschaften muss ein XML-Dokument erzeugt werden, das zum Beispiel folgenden Aufbau hat:

```
<toast duration="long">
  <visual version="1">
    <binding template="toastImageAndText02">
      <text id="1">Meine Kopfzeile</text>
      <text id="2">Hier wird ein Bild angezeigt!</text>
      <image id="1" src="bulbon.png" alt="Bild" />
    </binding>
  </visual>
  <audio silent="true" />
</toast>
```

Quellcode

```
...
using System.Xml.Linq;
using Windows.Data.Xml.Dom;
using Windows.UI.Notifications;
...
namespace DOKO
{
    class ToastNotificationFactory
    {
```

Interne Instanz der *ToastNotification* (für die Methode *Hide*) speichern:

```
        private ToastNotification _toast;
```

Unsere Schnittstelle nach außen:

```
        public string Kopfzeile {get; set;}
        public string Textzeile1 { get; set; }
        public string Textzeile2 { get; set; }
        public string Bild { get; set; }
        public string Parameter { get; set; }
        public bool LongRunning { get; set; }
        public bool SoundLoop { get; set; }
        public bool Silent { get; set; }
        public string Sound { get; set; }
```

HINWEIS: Alle Eigenschaften sind optional bzw. werden im Konstruktor initialisiert. Sie müssen mindestens eine Textzeile parametrieren. Die weiteren möglichen Varianten entnehmen Sie dem Abschnitt 21.3.2.

Der Standardkonstruktor zum Initialisieren der Werte:

```
        public ToastNotificationFactory()
        {
            Kopfzeile = "";
            Textzeile1 = "Hier muss ein Text rein!";
            Textzeile2 = "";
            Bild = "";
            Parameter = "";
```

```
            LongRunning = false;
            SoundLoop = false;
            Silent = false;
            Sound = "ms-winsoundevent:Notification.Looping.Alarm";
        }
```

Eine Überladung des Konstruktors mit optionalen Parametern erlaubt eine schnellere Definition der *ToastNotification*:

```
        public ToastNotificationFactory(string kopfzeile, string textzeile1 = "",
                                         string textzeile2 = "")
        {
            Kopfzeile = kopfzeile;
            Textzeile1 = textzeile1;
            Textzeile2 = textzeile2;
            Bild = "";
            Parameter = "";
            LongRunning = false;
            SoundLoop = false;
            Silent = false;
            Sound = "ms-winsoundevent:Notification.Looping.Alarm";
        }
```

Über diese Eigenschaft können Sie abfragen, ob Popup-Benachrichtigungen überhaupt angezeigt werden und wenn nein, warum:

```
        public NotificationSetting Setting
        {
            get
            {
                return ToastNotificationManager.CreateToastNotifier().Setting;
            }
        }
```

Hier haben wir es mit dem Herzstück der Klasse zu tun. Diese Methode erstellt in Abhängigkeit der vorgegebenen Parameter den korrekten XML-Code für die *ToastNotification*. Dazu ist es vor allem wichtig, welches Template gewählt wird (Text, Text+Überschrift, mit oder ohne Bild etc.). Die Programmlogik versucht dieses aus den Eigenschaften zu ermitteln.

```
        private string CreateXML()
        {
```

Falls eine Grafik vorhanden ist, ändert sich der Templatename:

```
            string templatename = "toast";
            if (Bild != "") templatename += "ImageAnd";
```

Mit Hilfe geschachtelter *XElement*-Konstruktoren erzeugen wir zunächst ein XML-Dokument:

```
            XElement xe = new XElement("toast", new XElement("visual",
               new XAttribute("version", "1"), new XElement("binding",
                  new XAttribute("template",""")) ));
```

Der XML-Code bisher:

```
<toast>
  <visual version="1">
    <binding template="" />
  </visual>
</toast>
```

Wir picken uns das Element *binding* heraus, um dieses einzeln weiterzubearbeiten:

```
var bind = xe.Element("visual").Element("binding");
```

Falls ein Parameter vorhanden ist, wird dieser als Attribut eingefügt:

```
if (Parameter != "")
    xe.Add(new XAttribute("launch", Parameter));
```

Gleiches gilt für die Laufzeit der Anzeige (7 oder 25 Sekunden):

```
if (LongRunning)
    xe.Add(new XAttribute("duration", "long"));
```

Die Tonwiedergabe wird über ein zusätzliches Element *audio* gesteuert:

```
if (SoundLoop & !Silent)
    xe.Add(new XElement("audio", new XAttribute("loop", "true"),
                new XAttribute("src", Sound)));
```

Falls Ruhe erwünscht ist, geht auch dies:

```
if (Silent)
    xe.Add(new XElement("audio", new XAttribute("silent", "true")));
```

Ist keine Kopfzeile definiert, wählen wir das passende Template und definieren ein Element für den Text der einzigen Zeile:

```
if (Kopfzeile == "")  // nur body1
{
    bind.Attribute("template").Value = templatename + "Text01";
    bind.Add(new XElement("text", new XAttribute("id", "1"), Textzeile1));
}
else
{
```

Hier folgen die alternativen Varianten (siehe Kommentare):

```
    if (Textzeile2 == "") // nur einzeiliger Bodytext
    {
        if (Kopfzeile.Length > 20) // lange Kopfzeile
        {
            bind.Attribute("template").Value = templatename + "Text03";
            bind.Add(new XElement("text", new XAttribute("id", "1"), Kopfzeile));
            bind.Add(new XElement("text", new XAttribute("id", "2"), Textzeile1));
        }
```

```
                else
                {
                    bind.Attribute("template").Value = templatename + "Text02";
                    bind.Add(new XElement("text", new XAttribute("id", "1"), Kopfzeile));
                    bind.Add(new XElement("text", new XAttribute("id", "2"), Textzeile1));
                }
            }
            else // zweizeiliger Bodytext
            {
                bind.Attribute("template").Value = templatename + "Text04";
                bind.Add(new XElement("text", new XAttribute("id", "1"), Kopfzeile));
                bind.Add(new XElement("text", new XAttribute("id", "2"), Textzeile1));
                bind.Add(new XElement("text", new XAttribute("id", "3"), Textzeile2));
            }
        }
```

Falls ein Bild definiert wurde, müssen wir dieses auch per Element einfügen:

```
        if (Bild != null)
            bind.Add(new XElement("image", new XAttribute("id", "1"),
                     new XAttribute("src", Bild), new XAttribute("alt", "Bild")));
```

Zurückgegeben wird der XML-Quellcode:

```
        return xe.ToString();
    }
```

Eine *ToastNotification* erzeugen:

```
    public ToastNotification CreateToast()
    {
        XmlDocument xdoc = new XmlDocument();
        xdoc.LoadXml(CreateXML());
        return new ToastNotification(xdoc);
    }
```

Eine *ScheduledToastNotification* erzeugen (wir benötigen eine Zeitangabe):

```
    public ScheduledToastNotification CreateScheduledToast(DateTimeOffset zeit)
    {
        XmlDocument xdoc = new XmlDocument();
        xdoc.LoadXml(CreateXML());
        return new ScheduledToastNotification(xdoc, zeit);
    }
```

HINWEIS: Die beiden obigen Methoden können Sie verwenden, wenn Sie das *ToastNotification*-Objekt noch weiter parametrieren wollen und direkt verwenden. Alternativ stehen Ihnen die Methoden *Show* und *AddToSchedule* zur Verfügung, die sich um die Anzeige kümmern:

```
        public ToastNotification Show()
        {
            _toast = CreateToast();
            ToastNotificationManager.CreateToastNotifier().Show(_toast);
            return _toast;
        }

        public void Hide()
        {
            if (_toast != null)
                ToastNotificationManager.CreateToastNotifier().Hide(_toast);
        }

        public ScheduledToastNotification AddToSchedule(DateTimeOffset zeit)
        {
            ScheduledToastNotification res = CreateScheduledToast(zeit);
            ToastNotificationManager.CreateToastNotifier().AddToSchedule(res);
            return res;
        }
    }
}
```

Test

HINWEIS: Siehe dazu Abschnitt 21.3.2 ab Seite 1089.

Platz für eigene Erweiterungen ist sicher immer vorhanden, aber 90 Prozent der Anwendungsfälle dürften durch unsere selbst entwickelte Klasse abgedeckt sein.

Anhang

Anhang A

Glossar

Begriff	Bedeutung	Bemerkung
ACE	Access Control Entries	Einträge in einer ACL
ACL	Access Control List	Zugangskontrollliste, dient der Rechteverwaltung
ADO	ActiveX Data Objects	ältere Datenzugriffstechnologie von Microsoft
ADO.-NET		neue Datenzugriffstechnologie von Microsoft für .NET
ADS	Active Directory Service	Verzeichnisdienst
ANSI	American National Standard Institute	US-amerikanische Standardisierungsbehörde
API	Application Programming Interface	allgemeine Schnittstelle für den Anwendungsprogrammierer
ASCII	American Standard Code for Information Interchange	klassisches Textformat
ASP	Active Server Pages	Webseiten mit serverseitig ausgeführten Skripten
BLOB	Binary Large Object	binäres Objekt, z.B. Grafik
BO	Business Object	Geschäftsobjekt
CAO	Client Activated Objects	vom Client aktiviertes Objekt (.NET Remoting)
CGI	Common Gateway Interface	Möglichkeit für die Verarbeitung von Anfragen auf einem Webserver
CLI	Common Language Infrastructure	Standard für alle .NET-Programmiersprachen
CLR	Common Language Runtime	virtuelle Umgebung von .NET
COD	Click Once Deployment	neue Distributionsmöglichkeit in .NET 2.0
COM	Common Object Model	allgemeines Objektmodell von Microsoft
CSV	Comma Separated Variables	durch bestimmte Zeichen getrennte Daten (meist Komma)
CTS	Common Type System	Datentypen, die von .NET unterstützt werden

Begriff	Bedeutung	Bemerkung
DAO	Data Access Objects	klassische Datenzugriffsobjekte
DC	Device Context	Gerätekontext
DCOM	Distributed Component Object Model	auf mehrere Rechner verteiltes COM
DES	Data Encryption Standard	Standard für die Verschlüsselung von Daten
DISCO	WebService Discovery	XML-Protokoll zum Aufsuchen von Webdiensten
DLL	Dynamic Link Library	Laufzeitbibliothek, die von mehreren Programmen benutzt werden kann
DQL	Data Query Language	Untermenge von SQL zur Datenabfrage
DDL	Data Definition Language	Untermenge von SQL zur Datendefinition
DML	Data Manipulation Language	Untermenge von SQL zur Datenmanipulation
DMO	Distributed Management Objects	Objekte z.B SQLDMO zum Administrieren des SQL Servers
DNS	Domain Name Service	Umwandlung von Domain-Namen in IP-Adresse
DOM	Document Object Model	objektorientiertes Modell für den Zugriff auf strukturierte Dokumente
DSN	Data Source Name	Name einer Datenquelle
DTD	Document Type Definition	Definition der XML-Dokumentenstruktur
DTS	Data Transformation Services	SQL-Server-Dienst, zum Transformieren von Daten
FCL	Framework Class Library	.NET-Klassenbibliothek
FSM	Finite State Machine	Endlicher Zustandsautomat
FTP	File Transfer Protocol	Internet-Protokoll für Dateitransfer
FQDN	Full Qualified Domain Name	Host-Name des Servers in URL
FSO	File System Objects	Objektmodell für Zugriff auf Laufwerke, Verzeichnisse und Dateien
GAC	Global Assembly Cache	allgemein zugänglicher Speicherbereich für Assemblies
GC	Garbage Collection	"Müllsammlung" (Freigabe von Objekten)
GDI	Graphical Device Interface	Grafikfunktionen der Windows API
GDI+		Grafikklassenbibliothek von .NET
GLS	Gleichungssystem	Begriff der numerischen Mathematik
GUI	Graphical User Interface	grafische Benutzerschnittstelle
GUID	Global Unique Identifier	eindeutiger Zufallswert (128 Bit) zur Kennzeichnung von Klassen
HTML	Hypertext Markup Language	Sprache zur Gestaltung statischer Webseiten
HTTP	Hypertext Transfer Protocol	Protokoll für Hypertextdokumente

Begriff	Bedeutung	Bemerkung
ICMP	Internet Control Message Protocol	Nachrichtenprotokoll im Internet
ID	Identifier	Identifikationsschlüssel
IDC	Internet Database Connector	... enthält Infos zum Herstellen einer Verbindung bzw. Ausführen von SQL
IDE	Integrated Development Environment	Integrierte Entwicklungsumgebung
IE	Internet Explorer	... oder Internet Browser
IIS	Internet Information Server	... oder Internet Information Services
IL	Intermediate Language	Zwischencode von .NET
ISAM	Indexed Sequence Access Method	indexsequenzielle Zugriffsmethode
ISAPI	Internet Server API Interface	Web-Anwendung (DLL) für IIS und IE
Jet	Joint Engineers Technology	lokales Datenbanksystem von Microsoft
JIT	Just In Time	Kompilieren zur Laufzeit
LAN	Local Area Network	lokales Rechnernetzwerk
MARS	Multiple Active Results Sets	Mehrfachverwendung einer Connection
MDA	Model Driven Architecture	Anwendungsentwicklung auf Basis von Modellen
MDAC	Microsoft Data Access Components	Datenzugriffskomponenten (ab Version 2.6), müssen auf Zielcomputer installiert sein
MIME	Multipurpose Internet Mail Extensions	standardisierte Dateitypen für Internet-Nachrichten
MMC	Microsoft Management Console	Rahmenanwendung für administrative Aufgaben
MS	Microsoft	Software-Gigant
MSDE	Microsoft Data Engine	abgerüstete SQL Server-Datenbank-Engine
MSDN	Microsoft Developers Network	eine (fast) unerschöpfliche Informationsquelle für den Windows-Programmierer
MSIL	Microsoft Intermediate Language	Zwischencode für .NET
MSXML	Microsoft XML Core Services	
ODBC	Open Database Connectivity	allgemeine Datenbankschnittstelle
OLAP	On-Line Analytical Processing	
OLE	Object Linking and Embedding	Microsoft-Technologie zum Verknüpfen und Einbetten von Objekten
OLE DB		Schnittstelle für den universellen Datenzugriff
OOP	Object Oriented Programming	Objektorientierte Programmierung
PAP	Programmablaufplan	

Begriff	Bedeutung	Bemerkung
POP3	Post Office Protocol Version 3	Posteingangsserver
PWS	Personal Web Server	abgerüstete Version des IIS
RAD	Rapid Application Development	schnelle Anwendungsentwicklung
RDBMS	Relational Database Management System	Relationales Datenbank-Management-System
RDL	Report Definition Language	XML-basierte Beschreibungssprache für Microsoft Reporting Services
RDS	Remote Data Services	Objektmodell für Datenverkehr mit Remote Server
RPC	Remote Procedure Call	Aufruf einer entfernten Methode
RTL	Runtime Library	Laufzeitbibliothek
SAO	Server Activated Object	vom Server aktiviertes Objekt (.NET Remoting)
SDK	Software Development Kit	Entwickler-Tools
SGML	Standard Generalized Markup Language	Regelwerk zur Definition von Auszeichnungssprachen für Dokumente
SMO	SQL Management Objects	managed Code-Libraries zur Verwaltung und Analyse des SQL Servers
SMTP	Simple Mail Transport Protocol	TCP/IP-Protokoll für die Übertragung von Nachrichten zwischen einzelnen Computern
SOAP	Simple Object Access Protocol	Protokoll zum XML-basierten Zugriff auf Objekte
SOM	Schema Object Model	zusätzliche APIs für den Zugriff auf XML Schema-Dokumente
SQL	Structured Query Language	Abfragesprache für Datenbanken
SQLDMO	SQL Distributed Management Objects	Library für Verwaltung des MS SQL Servers
SSL	Secure Socket Layer	Sicherheitsprotokoll für Datenübertragung
SSPI	Security Service Provider Interface	API für Authentifizierung und Vergabe von Zugriffsberechtigungen
TCP/IP	Transmission Control Protocol/ Internet Protocol	Netzwerkprotokoll zum Datentransfer, IP-Adresse ist 32-Bit-Zahl
UDDI	Universal Description, Discovery and Integration	Technologie zum Durchsuchen nach Webdiensten
UDF	User Defined Function	benutzerdefinierte Funktion (SQL Server)
UDL	Unified Data Link	standardisierte Datenverbindung
UDP	Unified Data Protocol	standardisiertes Datenprotokoll
UI	User Interface	Benutzerschnittstelle
UML	Unified Modelling Language	Sprache zur Beschreibung von Objektmodellen

Begriff	Bedeutung	Bemerkung
UNC	Uniform Naming Convention	System zur Benennung von Dateien in vernetzten Umgebungen
URL	Uniform Resource Locator	Web-Adresse
WMI	Windows Management Instrumentation	Klassen zur Windows-Administration
WinRT	Windows Runtime	Neues Framework seit Windows 8
WSDL	Web Services Description Language	XML-basierte Beschreibungssprache für Webdienste
WSE	Webservice Enhancements	Webdienst-Erweiterungen von Microsoft
W3C	Consortium	Standard
WWW	World Wide Web	Teil des Internets
XAML	eXtensible Application Markup Language	XML-Beschreibung für Windows-Oberflächen
XML	Extensible Markup Language	universelle textbasierte Beschreibungssprache
XSD	XML Schema Definition Language	XML-Dialekt zur Beschreibung von Datenstrukturen
XSLT	Extensible Stylesheet Language Transformations	Technologie zum Transformieren der Struktur von XML-Dokumenten

Anhang B

Wichtige Dateiextensions

Extension	Beschreibung
.ascx	Web-Benutzersteuerelemente
.asp	Active Server Pages
.aspx	Webform
.aspx.cs	Quellcode für Webform
.cd	vom Klassen Designer angelegte Datei
.config	Konfigurationsdatei der Anwendung
.csproj	C#-Projektdatei
.css	StyleSheet
.deploy	Dateien für Click Once Deployment
.disco	Static Discovery File
.dll	Assembly (Klassenbibliothek)
.exe	Assembly (ausführbare Datei)
htm	HTML-Datei
manifest	Deployment Manifest
.pdb	Debug-Infos (Program Debug Database)
resources	Ressourcen-Datei
resx	Ressourcen-Datei (XML)
rdl	XML-Report (Reporting Services)
rdlc	lokaler Xml-Report
rpt	Crystal Report
.settings	Anwendungseinstellungen (Visual Studio Settings)
.sln	Visual Studio Projektmappe
.suo	Benutzereinstellungen Visual Studio
.cs	C#-Quellcodedatei

Extension	Beschreibung
.vshost.exe	Visual Studio Host zum Laden der Assembly
.wsf	Script für Windows Scripting Host
.xsd	XML-Schema für XML-Dokumente
.xslt	XML-Transformationsdatei
default.aspx	Standardseite für Web
global.asax	Globale Ereignisse für die Webanwendung
web.config	WEB-Konfiguration
web.sitemap	Inhaltsverzeichnis des Webs für die Navigation

Index

??-Operator 113
.BAML 597
.G.CS 598
.NET WinRT-Profil 868
.NET-Framework 75

A

Abbruchbedingung 554
Abhängige Eigenschaften 737
Abort 441
Abs 278
abstract 209
Abstraktion 73
AcceptReturn 641
Access Control Entries 387
Access-Key 637
Accessor 174
ACE 387
ACL 387
Activated 624
ActualHeight 637
ActualWidth 637
AddAccessRule 387
AddDays 275
AddExtension 422
AddHours 275
AddMinutes 275
AddMonths 275
AddRange 306
AddYears 275
Aktivierung 1009
Alias 300
AllowsTransparency 627
Anfangswerte 107
Angehängte Eigenschaften 739
AngleX 774
AngleY 774
Animationen 777
Anonyme Methoden 319, 340
Anonyme Typen 348
Ansichtsänderung 926
Anwendungsdienste 890
Anwendungsmenü 729
App 589, 621
App-Ansichten 917
App-Leisten 903
App.config 562
App.Current 590
App.xaml 589, 997
App.xaml.cs 590
AppBar 985, 1103
AppData 1018
Append 407

AppendLinesAsync 1059
AppendText 409
AppendTextAsync 1059
ApplicationCommands 751
ApplicationDataCompositeValue 1024
ApplicationDataContainer 1023
ApplicationView 922
ApplyPropertyValue 683
Arithmetische Operatoren 122
ARM 871
Array 239, 250
ArrayList 305, 310, 314
as-Operator 116
Assemblierung 64, 79, 92
AssemblyInfo.cs 995
Assemblyinformation 996
Assets 908, 1006
async 475
Asynchrone Programmierentwurfsmuster 464
Atn 278
Attached Properties 603, 739
Attribute 80
Attributes 384, 1050
Auflistung 303
Aufzählungszeichen 674
Ausgabefenster 550
Ausnahmen 573
Auto-Play 1037
AutoGenerateColumns 825
AutomationProperties 1103
AutoPlay 963
AutoResetEvent 538
AutoToolTipPlacement 658
AutoToolTipPrecision 658
AveragePagesPerMinute 855
await 475

B

Background 616, 635, 681
BackgroundWorker 461
BandIndex 668
Barrier 533
base 198
Basic Page 912
Basisklassen 197
Beep 440
Befehlsdesign 902
Befehlsfenster 548
Begin 779
BeginAnimation 778
BeginInit 653
BeginInvoke 457, 471
BeginningEdit 829
Begrüßungsbildschirm 998
Benannte Styles 755
Benutzerschnittstelle 889
BigInteger 328
Binärdatei 411
Binary Application Markup Language 597
BinaryFormatter 406, 414
BinaryReader 406, 411, 412
BinarySearch 249
BinaryWriter 406, 411, 412
Binding 788
BindingBase.StringFormat 823
BindingNavigator 424
BindingSource 424
Bindungsarten 788
BitmapFrame 653
BitmapImage 653
BlackoutDates 704
Blank Page 911
BlockingCollection 533
Blocks 681
BlockUIContainer 676, 688

bool 105
BooleanNegationConverter 1003
BooleanToVisibilityConverter 1003
Boolesche Operatoren 125
Border 672, 946
BorderBrush 635
BorderThickness 672
BottomAppBar 1103
Boxing 120
break 133, 411
Breakpoints 552
BringIntoView 700
Browser 711
Bubbling Events 745
BulletDecorator 674
Button 638, 936
byte 105

C

C#-Compiler 60
C#-Source-Datei 62
Calendar 702
Callback 467
CallbackTimer 482, 486
Caller Information 564
CallerFilePath 564
CallerLineNumber 564
CallerMemberName 564
CancellationPending 462
CancellationToken 524, 537
CancellationTokenSource 524
CanExecute 754
Canvas 600, 606, 947
Canvas.Bottom 606
Canvas.Right 606
CaptureElement 961
CaretBrush 642
CaretIndex 641
case 128
CenterOwner 625
CenterScreen 625
CenterX 773
CenterY 773
char 105
CharmBar 1080
Chart 732
CheckAccess 459
CheckBox 643, 938
CheckFileExists 393, 422
CheckPathExits 393
class 64, 161
ClassLoader 78
Clear 247, 249
ClickOnce 583
Clone 249, 257, 258
Close 194
ClosedByUser 1010
Cloud 1025
CLR 75, 77
CLR-Threadpool 506
CLS 75
Code Contracts 578
Code Manager 78
Codefenster 72
Collapsed 636
Collation 857
Collection 303, 798
CollectionView 808
CollectionViewSource 801, 974, 981
ColumnDefinitions 612
ColumnSpan 616, 950
COM-Komponenten 79
COM-Marshaller 78
ComboBox 649, 951
Command 682, 749
Commands 748, 987
CommandTarget 682, 750

Common Language Runtime 72, 77
Common Language Specification 75, 76
Common Type System 75, 76
Completed 779
Complex 331
ComponentCommands 751
ConcurrentBag 533
ConcurrentDictionary 533
ConcurrentQueue 533
ConcurrentStack 533
const 111
Constraint 315
ContentPresenter 765
ContentsChanged 1126
ContextMenu 665
continue 131
Contract 866, 994
ControlTemplate 765
Convert 118, 821
ConvertBack 821
Copy 384
CopyAsync 1062
CopyCount 857
CopyTo 249, 257, 258, 384
CornerRadius 672
Cos 278
CountdownEvent 533
Create 408
CreateDirectory 381
CreateEncryptor 417
CreateFileAsync 1057
CreateFileQueryWithOptions 1055
CreateFolderAsync 1054
CreateFolderQueryWithOptions 1052
CreateFromFile 419
CreateNew 420
CreateSubdirectory 381
CreateText 408
CreateViewAccessor 420
CreateXpsDocumentWriter 857, 859
CreationCollisionOption 1021, 1054
CreationTime 384
Credentials 1028
CryptoStream 416, 417
CTS 75
Current 304
CurrentItem 801, 809
CurrentJobSettings 855
CurrentPosition 809
CurrentStateChanged 963, 989
Cursor 635

D

DataChanged 1027
DataFormats 679
DataGrid 701, 825
DataGridCheckBoxColumn 826
DataGridTextColumn 826
DataPackage 1073, 1080
DataPackageView 1073
DataRequested 1080
DataTemplate 803, 959
DataTime 274
DataTransfer 1073
DataTransferManager 1080
DateCreated 1050
Dateiattribute 384
Dateien kopieren und verschieben 383
Dateien umbenennen 384
Dateiparameter 407
Dateipicker 1062
Dateitypzuordnung 1043
Dateiverknüpfung 1043
Daten-Trigger 763
Datenstrukturen 532
Datentypen 104
Datentypzuordnung 1037

Datenzugriff 136
DatePicker 702
DateTime 274
Datumsformatierung 282
Datumsfunktionen 273
Day 274
DayOfWeek 274
DayOfYear 274
DaysInMonth 275
Deactivated 624
Deadlocks 437
Debug 557
  Write 558
  WriteIf 558
  WriteLineIf 558
Debuggen 1014
Debugger 547
DebugView 1016
decimal 105
DecodePixelWidth 653
Decrypt 415
default 128, 131
DefaultExt 422
DefaultFileExtension 1066
DefaultPrintQueue 855
DefaultPrintTicket 855
DefaultView 801
Deklarationen 1001
Dekrement 122
Delay 791
Delegate 182, 315, 340
Delegate instanziieren 318
Delete 381
DeleteAsync 1054, 1062
DeleteContainer 1028
Dependency Properties 737
DependencyObject 738
DereferenceLinks 393
DESCryptoServiceProvider 417
Designer 71
DesktopDirectory 394
Destruktor 188, 191
DeviceFontSubstitution 857
DeviceInformation 897, 961
DeviceInformationCollection 961
Diagnostics 439
DialogResult 422
Dictionary 314
Dimensionsgrenzen 244
Directory 376, 380
DirectoryInfo 376
DirectoryName 380
DirectorySecurity 387
DirectX 583
Direkte Events 747
Diskpart 872
DispatcherUnhandledException 624
DisplayDate 702
DisplayMemberPath 802
DisplayMode 702
DisplayType 1050
do 131, 132
DockPanel 600, 608
DockPanel.Dock 608
Document 678
DocumentPaginator 844, 849
DocumentsLibrary 1023
DocumentViewer 691
double 105
DownloadsFolder 1022
DragItemsStarting 955
DriveInfo 376
Druckerauswahl 853
Druckerinfos 853
Duplexing 857
DynamicResource 742
Dynamische Programmierung 324

E

EditingCommands 683, 751
EditingMode 707
Eigenschaften 66, 170
Eigenschaften-Fenster 71
Eigenschaften-Trigger 760
Einzelschritt-Modus 555
Element 1136
Ellipse 710
else if 128
EnableRaisingEvents 390
Encrypt 415
EndInvoke 457, 471
EndsWith 252
Enter 449
Entity Data Model 833
Entwicklungsumgebung 68
enum 134
EraseByPoint 707
EraseByStroke 707
Ereignis 181, 182
Ereignis auslösen 184
Ereignis-Trigger 762
Ereignismethoden 752
Ereignisse 67, 181
Erweiterungsmethoden 350, 367
event 182
EventLog 563
EventLogTraceListener 563
Events 67
Exception 574
ExceptionManager 78
Exists 385
Exit 449, 624
Exp 278
ExpandDirection 693
Expander 692
Exponentialfunktion 280
eXtensible Application Markup Language 585
Extension 380, 995
Extension Method-Syntax 352, 367

F

FailIfExists 1022
Fast and Fluid 865
Fehlerbehandlung 565
Fehlerklassen 567, 575
File 376, 406
File Type Association 1043
FileAccess 407
FileInfo 376, 406, 409
FileIO 1058, 1060
FileMode 407
FileName 392, 422
FileOpenPicker 1062
FileSavePicker 1062, 1066
FileSecurity 387
FileShare 408
FileStream 406
FileSystemAccessRule 387
FileSystemWatcher 376, 390
FileTypeChoices 1066
Fill 617
Filled 918, 919
Filter 390, 422, 810
FilterIndex 392
Filters 392
FindAllAsync 897
FindResource 793
Fingereingabe 905
FitToCurve 706
Fix-Dokumente 844
FixedPrintManager 848
FlipView 959
float 105
FlowDirection 607

FlowDocument 676, 678, 687
FlowDocumentPageViewer 686
FlowDocumentReader 687
FlowDocumentScrollViewer 687
FlowPrintManager 850
Flyout 1097
FolderBrowserDialog 393
FolderDepth 1055, 1056
FolderPicker 1062, 1067
FolderRelativeId 1050
FontFamily 635
Fonts 394
FontSize 635
FontStyle 635
FontWeight 635
for 131, 132
foreach 207, 242, 314
Foreground 635
Form1.cs 71
Format 283, 820
Formulare 66
Frame 901, 906, 964
Freigabeziel 1083, 1128
FromCurrentSynchronizationContext 529, 544
Füllmodus 918
FullName 380
FullScreenLandscape 919
FullScreenPortrait 919
Funktionen 1000
FutureAccessList 1069

G

Garbage Collector 191
GenerateUniqueName 1021, 1054
Generics 309, 311
Gestensteuerung 943
GestureOnly 707
get 170, 174
GetBasicPropertiesAsync 1050
GetBitmapAsync 1073
GetCreationTime 385
GetCurrentDirectory 382
GetDataAsync 1073
GetDefaultView 800, 981
GetDirectories 382
GetEnumerator 304, 313
GetExecutingAssembly 996
GetFiles 386
GetFilesAsync 1055
GetFoldersAsync 1052
GetLength 249, 258
GetPrintQueues 854
GetProcessById 497
GetProcesses 496
GetStorageItemsAsync 1073
GetTypeInfo 996
GoBack 928
goto 131
GPS 890
Grafikausgabe 583
Grafikskalierung 655
Grid 600, 612, 949
Grid App 910
Grid.Column 614
Grid.Row 614
GridSplitter 616
GridView 701, 956
GridView gruppieren 972
GroupBox 673
GroupName 645
Grundlayout 914

H

Haltepunkte 554
Hardwarebeschleunigung 583
Hashtable 307

HasPaperProblem 856
HasValue 113
Header 693
HeaderTemplate 973
Hidden 636
HomeGroup 1023
HorizontalAlignment 605, 615, 635
HorizontalContentAlignment 636
HorizontalOffset 695
HorizontalScrollBarVisibility 660
HostingPrintServer 856
Hour 274
Hover 936
HTML 5 900
HtmlFormatHelper 1074
HyperlinkButton 936

I

IAsyncResult 466, 467, 468, 473
ICollection 304
Icon 626
Idle-Prozesse 496
IEnumerable 303, 311
IEnumerator 304
if 128
IInspectable 892
ILSpy 869
Image 652, 943
IndentLevel 559
IndentSize 559
Index 239
Indexer 302, 309, 338
IndexerOption 1056
IndexOf 249, 252
Indexprüfung 241
InitialDirectory 422
Initialisierung 163
Initialize 249, 258
InitializeComponent 590
InkAndGesture 707
InkCanvas 706
Inkrement 122
InlineUIContainer 685
INotifyCollectionChanged 798
INotifyPropertyChanged 795
InputGestureText 662
InputScopeName 970
Insert 252
InsertBefore 681
InsertParagraphBreak 681
InsertTextInRun 681
Installationsverzeichnis 1021
InstalledLocation 1021
Instanz 158
Instanziieren 162
int 105
Int16 105
Int32 105
Int64 105
Intellisense 167
IntelliTrace 536
internal 159
internal protected 160
Interrupt 441
InvalidateVisualState 920
InvalidOperationException 802
Invoke 456, 469, 471
InvokeRequired 458
IsActive 945
IsAlive 442
IsBackGround 443
IsCheckable 664
IsChecked 643, 645, 664
IsCompleted 466, 512
IsCurrentAfterLast 801, 809
IsCurrentBeforeFirst 801, 809
IsDirectionReversed 658

IsEditable 649
IsExpanded 694, 700
IsIndeterminate 671
IsLeapYear 275
IsLocked 668
IsLooping 963
IsManualFeedRequired 856
IsMuted 656, 963
IsOpen 696
IsOutOfPaper 856
IsReadOnly 641
IsSelected 700
IsSelectionRangeEnabled 658
IsSnapToTickEnabled 659
IsSourceGrouped 972
IsSynchronizedWithCurrentItem 802
IsThreeState 643
ItemsControl 951
ItemsPage 908
ItemsPanelTemplate 973
ItemTemplate 909
Iterator 513
Iteratoren 313
IUnknown 892
IValueConverter 821

J

JavaScript 900
JIT-Compiler 74
Join 441

K

Kapselung 73, 158
Kind-Elemente 614
Klasse 158
KnownFolders 1023
Kommentare 103
Komprimieren 417
Konsolenanwendung 145
Konstante Felder 176
Konstanten 104, 111
Konstruktor 188, 495
Kontaktliste 1084
Kontravarianz 328
Kovarianz 328
kritische Abschnitte 482
Kurz-Operatoren 123
Kurzschlussauswertung 126

L

Label 637
Lambda Expression 340
Lambda-Ausdrücke 320, 367
Language Projection 868, 892
LastAccessTime 384
LastChildFill 609
LastWriteTime 385
Laufwerke 382
Launch-Contract 1009
Layout 600
Layout-Control 947
LayoutAwarePage 906, 1005
LayoutTransform 709
Lebenszyklus 1007
Leere Seite 911
Leerzeichen 620
Length 249, 252, 257, 258
Line 711
LineBreak 620
LINQ
  Abfrage-Operatoren 353
  Aggregat-Operatoren 361
  AsEnumerable 364
  Count 361
  GroupBy 358

Grundlagen 347
Gruppierungsoperator 358
Join 361
Konvertierungsmethoden 364
OrderBy 357
OrderByDescending 357
Projektionsoperatoren 355
Restriktionsoperator 356
Reverse 358
Select 355
SelectMany 355
Sortierungsoperatoren 357
Sum 362
ThenBy 357
ToArray 364
ToDictionary 364
ToList 364
ToLookup 364
Where 356
LINQ-Abfrageoperatoren 351
LINQ-Architektur 347
LINQ-Provider 348
LINQ-Syntax 351
List 309, 314, 676, 688
List Klasse 314
ListBox 646, 951
ListView 700, 806, 955
LoadCompleted 965
LoadedBehavior 656
LoadState 1004
LocalApplicationData 394
LocalFolder 1019
LocalPrintServer 854
LocalState 1019
lock 447
Log 278
Log10 278
Logarithmus 280
LogicalDpi 923
Logische Operatoren 124
Lokal-Fenster 549
Lokale Variablen 114
long 105
LongRunning 528
LowestBreakIteration 512

M

MainModule 496
MainWindow.xaml 591
MainWindow.xaml.cs 592
ManipulationDelta 943
ManipulationMode 943
ManualResetEvent 541
ManualResetEventSlim 533
Map View 420
Margin 604
MatrixTransform 772
Max 279
MaxHeight 603, 636
MaximumRowsOrColumns 950
MaxLength 641
MaxLines 641
MaxWidth 603, 636
Media 439
MediaCapture 961
MediaCommands 751
MediaElement 656, 962
MediaEnded 963
MediaOpened 963
Memory Mapped File 418, 431
MemoryMappedFile 419
Menu 661
Menü
Grafiken 663
Tastenkürzel 662
MenuItem 661
Menüleiste 661

MessageDialog 987, 1086
Messwertliste 370
Metadaten 79, 869
Methoden 66, 137, 178
  generische 312
  überladen 151
Methodenzeiger 316
MethodImpl 454
Methods 66
Microsoft Expression Blend 781
Microsoft Intermediate Language Code 74
Min 279, 431
MinHeight 603, 636
MinLines 641
Minute 274
MinWidth 603, 636
MMF 418
Modale Dialoge 638
Monitor 449
Month 274
MostRecentlyUsedList 1071
Move 381, 384
MoveAndReplaceAsync 1062
MoveAsync 1062
MoveCurrentTo 809
MoveCurrentToFirst 801, 809
MoveCurrentToLast 801, 809
MoveCurrentToNext 801, 809
MoveCurrentToPosition 809
MoveCurrentToPrevious 801, 809
MoveNext 304
MoveTo 384
MRU-Liste 1072
MSIL-Code 74, 92
MultipleRange 703
Multiselect 393
Multitasking 436
Multithreading 81, 436, 482
MusicLibrary 1023
Musikwiedergabe 988
Mutex 452
MyComputer 394
MyDocuments 394

N

Name-Attribut 593
Namespace 78, 162
NaturalDuration 656
NaturalVideoHeight 656
NaturalVideoWidth 656
Navigate 907, 915
NavigateToString 964
Navigation 915
NavigationCacheMode 917
NavigationCommands 751
Navigationsdesign 904
new 188, 240
Notification 889
NotifyFilter 390
NotRunning 1010
Now 275
NTFS 416
null 112, 164
Nullable Type 112

O

object 105, 111
Objekt 157
Objektbaum 424
Objektinitialisierer 190, 219, 349, 350
ObservableCollection 799
ODER 126
OnActivated 1010
OnCachedFileUpdaterActivated 1010
OneTime 789
OneWay 789

OneWayToSource 789
OnExplicitShutdown 624
OnFileActivated 1010, 1037, 1039
OnFileOpenPickerActivated 1010
OnFileSavePickerActivated 1010
OnLastWindowClose 624
OnLaunched 906, 915, 997, 1009
OnMainWindowClose 624
OnNavigatedTo 916
OnSeachActivated 1010
OnShareTargetActivated 1010, 1083
OnStartUp 623
OnSuspending 997
OOP 155
Opacity 627, 636
Open 408
OpenFileDialog 391, 421, 654
OpenOrCreate 408
OpenText 410
Operatoren 121
Optionale Parameter 327
Orientation 606, 658, 659
out 141
OutputColor 857
OutputQuality 857
OverflowMode 669
override 198
OverwritePrompt 422

P

Package.appxmanifest 998
Padding 604, 636
PadLeft 252
PadRight 252
Page 901
PageBorderless 842, 858
PageMediaSize 842, 858
PageOrder 858
PageOrientation 842
PageScalingFactor 858
PAP 145
Paragraph 676, 688, 936
Parallel Diagnostic Tools 534
Parallel LINQ 533
Parallel-Programmierung 503
Parallel.For 509
Parallel.ForEach 513
Parallel.Invoke 507
Parallele Aufgaben 534
Parallele Stacks 535
ParallelLoopResult 512
Parameterübergabe 140, 141, 915, 924
Parent 636
Parse 118, 275
PasswordBox 640, 642, 939
PasswordVault 1028
Path 376, 389
Pause 779, 963
PeekChar 412
PhotoPrintingIntent 858
Pi 279
PickSaveFileAsync 1066
PicturesLibrary 1023
Placement 695
PlacementRectangle 696
PlacementTarget 696, 965
Play 963
PlayPauseTogglePressed 992
PLINQ 364, 533
Polling 465
Polymorphes Verhalten 205
Polymorphie 73, 159, 195, 207
PopUp 695, 1097
Popup-Benachrichtigungen 1089
PopupMenu 1101
Position 963
Potenz 279

Pow 279
PreferFairness 528
PresentationHost 586
Press 936
PreviousExecutionState 1010
PrintableAreaHeight 842
PrintableAreaWidth 842
PrintDialog 841
PrintQueue 854
PrintTicket 842
Priority 442
private 159
Procedure-Step 552
Process 495, 500
Process Sandboxing 866
Process.Start 501
ProcessName 496
ProcessThread 495
Programm starten 500
ProgressBar 671, 945
ProgressChanged 463
ProgressRing 945
Projektmappen-Explorer 70
Projekttyp 69
Properties 66
Property 216
Property-Accessoren 174
PropertyChanged 790
PropertyChangedEventHandler 878
protected 160
Protokolldatei 1014
Prozedurschritt 556
Prozesse 495
public 159
Pulse 449, 450
PulseAll 449, 450

Q

Query Expression-Syntax 352, 367
QueryOptions 1052, 1055
Queue 311, 314
QueueUserWorkItem 444

R

Racing 437
RadioButton 645, 938
Rahmenbreite 672
Random 280
RandomAccessStreamReference 1075
Rank 249, 257, 258
Raster-App 910
Read 407
ReadAllBytes 412
ReadAllLines 410
ReadAllText 410
ReadBufferAsync 1059
ReadLine 64
ReadLines 410
ReadLinesAsync 1059
ReadTextAsync 1059
ReadToEnd 410
ReadWrite 407
Rechtschreibkontrolle 686
Rectangle 710
ref 140, 141
Referenzieren 162
Referenztyp 111
Referenztypen 251
Reflection 996
Reflexion 79
Regex 971
Registerkarte 717
Release 936
ReleaseMutex 453

Remote-Debugging 875
RemovableDevices 1023
Remove 252
RemoveAccessRule 387
RenderTargetBitmap 708
RepeatButton 638, 936
Replace 252
RequestedTheme 931
Reset 304
ResolutionScale 923
Resources 636
Ressourcen 739
Resume 441, 779
Resuming 1012
RetrievePropertiesAsync 1051
return 131, 313
Ribbon 714
RibbonApplicationMenuItem 729
RibbonApplicationSplitMenuItem 723
RibbonGallery 723
RibbonMenuButton 727
RibbonQuickAccessToolBar 725
RibbonSplitMenuButton 723
RibbonSplitMenuItem 723
RibbonTextBox 724
RibbonWindow 726
RichEditBox 939
RichTextBlock 933
RichTextBox 676
Roaming 1025
Roamingdaten 1028
RoamingFolder 1020
RoamingStorageQuota 1020
RotateTransform 772
Round 279
Routed Events 745
RowDefinitions 612
RowDetailsTemplate 828
RowDetailsVisibilityMode 828, 829
RowSpan 616, 950
Rückrufmethode 465
Rücksprung 556
Run 936
Runden 279
Running 1007

S

Sandboxing 864
SaveFileDialog 391, 421
SaveState 1004
ScaleTransform 772
ScaleX 773
ScaleY 773
Schaltjahr 275
ScheduledToastNotification 1137
Schleifenabbruch 511
Schleifenanweisungen 131
Schlüsselwörter 103
Schriftarten 906
ScrollBar 659, 945
ScrollViewer 660, 948
sealed 210
Search-Contract 1030
SearchPage 1031
SearchPane 1032
Second 274
Section 676, 688
Security Engine 78
Seek 779
Seitennavigation 924
Seitenstatus 916
Seitentemplates 906
SelectedDate 702
SelectedItem 700
SelectedItemChanged 700
SelectedItems 647
Selection 681

SelectionBrush 642
SelectionMode 646
SemanticZoom 977
Semaphore 454
SemaphoreSlim 533
Separator 661
Sequenzielle Datei 413
Serialisieren 414
Serialisierung 81
Serializable 414
Serializable-Attribut 425
SessionEnding 624
set 170, 174
SetAccessControl 388
SetBitmap 1073, 1075
SetCurrentDirectory 382
SetData 1073
SetSource 963
SetStorageItems 1073, 1077
Settingsbereich 904
ShareOperation 1130
short 105
Show 627
ShowAsync 1086
ShowDialog 422, 627
ShowForSelectionAsync 1101
ShowShareUI 1082
Shutdown 624
Sign 279
Silverlight 584
Sin 279
Single-Step 552
SingleRange 703
SizeChanged 918, 919, 920
Skalieren mit ScaleTransform 773
SkewTransform 772
Sleep 441
Slider 658, 945
Snapped 918, 919
SnapsTo 945
Sort 249
SortDescriptions 810
SortedList 314
SortedSet 332
Source 652
SpecialFolder 393
SpeedRatio 656
SpellCheck.IsEnabled 641
Sperrmechanismen 445
SpinLock 533
SpinWait 533
Splash Screen 998, 1011, 1041
Split 252
Split App 908
SplitPage 908
SQLite 1108
SQLiteConnection 1112
Sqr 279
Stack 311
StackPanel 600, 606, 948
stand-alone XAML 586
StandardDataFormats 1131
Standarddrucker 855
StandardStyles.xaml 1005, 1103
StartInfo 500
StartPreviewAsync 962
Startseite 915
StartsWidth 252
Startup 623, 624
StartupEventArgs 623
StartupUri 589, 622
static 212
StaticResource 742, 758
Statische Klassen 212
Statische Methoden 179
Statischer Konstruktor 190
StatusBar 669
StatusBarItems 670

Statusmitteilung 1091
StepValue 945
Steuerelemente 66
Stop 779, 963
Stopped 1007
Storage-Interface 889
StorageApplicationPermissions 1069
StorageDeleteOption 1055
StorageFile 1057, 1061
StorageFileQueryResult 1126
StorageFolder 1049
Store 870
Storyboard 777, 913, 920
StreamReader 406, 410, 1060
StreamWriter 406, 409, 1059
Stretch 655
StretchDirection 655
String 105, 251
Stringaddition 288
StringFormatConverter 967
StringReader 406
StringWriter 406
struct 135, 413
Strukturvariable 136
Style 636, 757, 1006
Style anpassen 758, 759
Style ersetzen 758
Styles vererben 759
StylusTip 706
Subklassen 198, 199
SubString 252
SuggestedStartLocation 1068
Suspend 441
Suspended 1010
Suspending 1007, 1011
SuspendingDeferral 1012
SuspensionManager 883, 1004
SVG 943
switch 128
Symbole 930
System 105, 394
System-Ressourcen 744
System.Collections.Concurrent 533
System.Diagnostics 495
System.IO.Compression 417
System.IO.FileStream 405
System.IO.Stream 405
System.Nullable 112
System.Object 208
System.Printing 840
System.Security.AccessControl 388
System.Security.Cryptography 416
System.Threading 439, 506
System.Threading.Tasks 506
System.TimeSpan 496
System.Windows.Xps 841
SystemParameters 744

T

TabControl 694
TabIndex 636
Table 676, 688
Tablet 894
Tablet-Hardware 895
Tablet-Simulator 944
TabPanel 600
Tag 636
Tan 279
Target 637
Task
  Canceled 528
  ContinueWith 520, 529
  Created 528
  Datenübergabe 517
  Faulted 528
  Fehlerbehandlung 526
  IsCanceled 528

IsCompleted 528
IsFaulted 528
Klasse 514
RanToCompletion 528
Result 520
return 523
Rückgabewerte 520
Running 528
Start 516
Starten 515
Status 528
Task-Ende 529
Task-Id 527
TaskCreationOptions 528
User-Interface 528
Verarbeitung abbrechen 523
Wait 518
WaitAll 519
WaitingForActivation 528
WaitingForChildrenToComplete 528
WaitingToRun 528
Weitere Eigenschaften 527
Task.Factory.StartNew 514, 539
TaskCreationOptions 538
TaskScheduler 529, 537
Tasteneingaben 971
Template 764
TemporaryFolder 1020
TemporaryKey.pfx 1002
Terminated 1010
Text 650
Textausrichtung 621
TextBlock 618, 933
TextBox 640, 939
Textdatei 408, 421
Texte formatieren 682
Textformatierungen 619
TextFormattingMode 630
TextPointer 681
TextRange 678, 680
TextWriterTraceListener 561
Thin Client 221
Thread 439, 440, 482
initialisieren 482
synchronisieren 482
ThreadState 497
Thread Service 78
Thread<> 479
Threadingmodell 591
ThreadInteruptedException 441
ThreadPool 443
Threads 495
threadsicher 456
Threadsichere Collections 532
ThreadState 442
ThreadWaitReason 498
Throw 568, 575
ThrowIfCancellationRequested 524
Thumbnail-Ansicht 1064
TickFrequency 658
TickPlacement 658
TickValues 945
Tiles 864
Timer-Threads 460
TimeSpan 496
TimeSpan-Klasse 288
Title 392, 422, 626
ToastNotificationFactory 1090
ToastNotificationManager 1135, 1138
ToastNotifications 1133
ToCharArray 252
Today 275
ToggleButton 638, 682, 938
ToggleSwitch 938
ToLongDateString 275
ToLongTimeString 275
ToLower 252
ToolBar 666

ToolBarTray 666, 667
Toolbox 71
ToolTip 636, 965
ToolTipPlacement 965
ToolTipService 965
TopAppBar 1103
ToShortDateString 275
ToShortTimeString 275
ToString 117, 281
TotalProcessorTime 496
Touchscreen 884
ToUpper 252
TPL 506
Trace 557, 561
TraceListener 561
TrackBar 288
Transformationen 771
TransformGroup 774
TranslateTransform 772
Transparenz 627
TreeView 398, 697
Trefferanzahl 555
Trigger 760
Trim 252
TrueTypeFontMode 858
Truncate 408
Try-Catch 566
Try-Finally 571
TryEnter 449, 452
TryUnsnap 922
Tunneling Events 745
Tuple 331
TwoWay 789
Typ-Styles 756
Typecasting 310, 337
Typinferenz 113, 349, 367
Typsicherheit 310
Typsuffixe 108

U

Überladene Methoden 179
Überwachungsfenster 550
Uhr anzeigen 276
UI-Virtualisierung 826
UICommand 987, 1101
UML 195
Unboxing 120
UND 126
Unicode 109
Unified Modeling Language 195
Uniform 617
UniformGrid 600, 610
UniformToFill 617
UnIndent 559
Unterverzeichnis 382
UpdateSourceTrigger 790
Uri 653, 743
UriSource 653
UseIndexerWhenAvailable 1056
UserPrintTicket 857
UserProfile 394
using 64, 162, 299, 431

V

ValidateNames 422
Value 658
var 107, 113
Variablen 104
Variablentypen 104
VariableSizedWrapGrid 950, 975
Verarbeitungsstatus 512
Vererbung 158
Verformen mit SkewTransform 773
Vergleichsoperatoren 124
Verpacken 1002, 1118
Verschieben mit TranslateTransform 774

Verschlüsseln 415
VerticalAlignment 605, 615, 635
VerticalContentAlignment 636
VerticalOffset 695
VerticalScrollBarVisibility 660
Vertrieb 1118
Verweistypen 106
Verzögerte Initialisierung 194
VideoCapture 961
VideosLibrary 1023
ViewBox 600, 617
ViewManagement 919
ViewMode 1064
ViewStateChanged 920
VirtualizingStackPanel 826
Virtuelle Tastatur 969
Visibility 636
Visual Studio 2012 Express 57
Visual Studio 2012 Premium 59
Visual Studio 2012 Professional 58
Visual Studio 2012 Ultimate 59
VisualStateManager 912, 919, 920
void 139
Vollbildmodus 870
Volume 963

W

Wait 449, 450
WaitForExit 500
WaitOne 453, 454, 539, 543
Webcam 890
WebView 964
WebViewBrush 965
Wechselspeichergeräte 1037
Werkzeugkasten 71
Wertetypen 106
where 315
while 131, 132
Wiederverwendbarkeit 158
Window 901
Windows 8 872
Windows App Certification Kit 1120
Windows Presentation Foundation 581
Windows Runtime 863
Windows Store 870
Windows Store App 864, 993
Windows-Simulator 873
WindowsFormsHost 733
WindowStartupLocation 625
WindowStyle 626, 627
Winkel 279
WinMD 893
WinMD-Dateien 869
WinRT 863, 887
WinRT-API 888
WinRT-COM 868
WinRT-Namespaces 890
work stealing 507
WorkerReportsProgress 463
WPF 459, 581
  Anwendung beenden 624
  Applikationstypen 596
  Eigenschaften 635
  Ereignis-Handler 592
  Ereignis-Modell 744
  Height 603
  Kommandozeilenparameter 623
  Left 603
  Maßangaben 602
  Startobjekt festlegen 622
  Style-System 754
  Top 603
  Width 603
  Window-Klasse 625
  Zielplattformen 595
WPF Toolkit 732
WPF-Programm 621

WPF-Wertkonvertierer 821
Wrap 619
WrapPanel 600, 610
WrapWithOverflow 619
Write 407
WriteAllBytes 412
WriteBufferAsync 1059
WriteIf 557
WriteLine 64, 557
WriteTextAsync 1059
Wurzel 279

X

x:Class 591
XAML 585, 900
XAML-Anwendung 586
XAML-Editor 589
XAttribute 1135
XCopy-Deployment 583
XElement 1135
XML Paper Specification 839
xml:space="preserve" 620
XmlDocument 1137
XOR 126
XPS-Dokumente 839
XpsDocumentWriter 859

Y

Year 274
yield 313, 340

Z

Zahlenformatierung 282
Zeilenumbrüche 620
Zeitfunktionen 273
Zeitmessung 289
Zielplattformen 871
ZoomedInView 977
ZoomedOutView 977
Zufallszahlen 280
Zugriffsberechtigung 387
Zuweisungsoperatoren 123
Zwischenablage 1073

?

??-Operator 113

.

.BAML 597
.G.CS 598
.NET WinRT-Profil 868
.NET-Framework 75

Walter Doberenz
Thomas Gewinnus

Ergänzungsband Visual C# 2012

Grundlagen und Profiwissen

HANSER

Die Autoren:

Professor Dr.-Ing. habil. Walter Doberenz, Wintersdorf

Dipl.-Ing. Thomas Gewinnus, Frankfurt/Oder

Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <*http://dnb.d-nb.de*> abrufbar.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdrucks und der Vervielfältigung des Buches, oder Teilen daraus, sind vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2013 Carl Hanser Verlag München
http://www.hanser-fachbuch.de

Lektorat: Sieglinde Schärl
Satz: Ingenieurbüro Gewinnus
Sprachlektorat: Walter Doberenz

ISBN 978-3-446-43439-4

E-Book-ISBN 978-3-446-43521-6

Inhaltsverzeichnis

Zweites Vorwort **1193**

Teil V: Weitere Technologien

22 XML in Theorie und Praxis **1197**

22.1 XML – etwas Theorie 1197
22.1.1 Übersicht 1197
22.1.2 Der XML-Grundaufbau 1200
22.1.3 Wohlgeformte Dokumente 1201
22.1.4 Processing Instructions (PI) 1204
22.1.5 Elemente und Attribute 1204
22.1.6 Verwendbare Zeichensätze 1206
22.2 XSD-Schemas 1208
22.2.1 XSD-Schemas und ADO.NET 1208
22.2.2 XML-Schemas in Visual Studio analysieren 1210
22.2.3 XML-Datei mit XSD-Schema erzeugen 1213
22.2.4 XSD-Schema aus einer XML-Datei erzeugen 1214
22.3 Verwendung des DOM unter .NET 1215
22.3.1 Übersicht 1215
22.3.2 DOM-Integration in C# 1216
22.3.3 Laden von Dokumenten 1216
22.3.4 Erzeugen von XML-Dokumenten 1217
22.3.5 Auslesen von XML-Dateien 1219
22.3.6 Direktzugriff auf einzelne Elemente 1221
22.3.7 Einfügen von Informationen 1221
22.3.8 Suchen in den Baumzweigen 1224
22.4 XML-Verarbeitung mit LINQ to XML 1227
22.4.1 Die LINQ to XML-API 1227
22.4.2 Neue XML-Dokumente erzeugen 1229

22.4.3 Laden und Sichern von XML-Dokumenten ... 1231
22.4.4 Navigieren in XML-Daten ... 1232
22.4.5 Auswählen und Filtern ... 1234
22.4.6 Manipulieren der XML-Daten ... 1234
22.4.7 XML-Dokumente transformieren ... 1236
22.5 Weitere Möglichkeiten der XML-Verarbeitung ... 1239
22.5.1 Die relationale Sicht mit XmlDataDocument ... 1239
22.5.2 XML-Daten aus Objektstrukturen erzeugen ... 1242
22.5.3 Schnelles Suchen in XML-Daten mit XPathNavigator ... 1245
22.5.4 Schnelles Auslesen von XML-Daten mit XmlReader ... 1248
22.5.5 Erzeugen von XML-Daten mit XmlWriter ... 1250
22.5.6 XML transformieren mit XSLT ... 1252
22.6 Praxisbeispiele ... 1254
22.6.1 Mit dem DOM in XML-Dokumenten navigieren ... 1254
22.6.2 XML-Daten in eine TreeView einlesen ... 1257

23 Einführung in ADO.NET ... 1261

23.1 Eine kleine Übersicht ... 1261
23.1.1 Die ADO.NET-Klassenhierarchie ... 1261
23.1.2 Die Klassen der Datenprovider ... 1262
23.1.3 Das Zusammenspiel der ADO.NET-Klassen ... 1265
23.2 Das Connection-Objekt ... 1266
23.2.1 Allgemeiner Aufbau ... 1266
23.2.2 OleDbConnection ... 1266
23.2.3 Schließen einer Verbindung ... 1268
23.2.4 Eigenschaften des Connection-Objekts ... 1268
23.2.5 Methoden des Connection-Objekts ... 1270
23.2.6 Der ConnectionStringBuilder ... 1271
23.3 Das Command-Objekt ... 1271
23.3.1 Erzeugen und Anwenden eines Command-Objekts ... 1272
23.3.2 Erzeugen mittels CreateCommand-Methode ... 1272
23.3.3 Eigenschaften des Command-Objekts ... 1273
23.3.4 Methoden des Command-Objekts ... 1275
23.3.5 Freigabe von Connection- und Command-Objekten ... 1276
23.4 Parameter-Objekte ... 1277
23.4.1 Erzeugen und Anwenden eines Parameter-Objekts ... 1277
23.4.2 Eigenschaften des Parameter-Objekts ... 1278

23.5 Das CommandBuilder-Objekt ... 1279
23.5.1 Erzeugen ... 1279
23.5.2 Anwenden ... 1279
23.6 Das DataReader-Objekt ... 1280
23.6.1 DataReader erzeugen ... 1280
23.6.2 Daten lesen ... 1281
23.6.3 Eigenschaften des DataReaders ... 1282
23.6.4 Methoden des DataReaders ... 1282
23.7 Das DataAdapter-Objekt ... 1283
23.7.1 DataAdapter erzeugen ... 1283
23.7.2 Command-Eigenschaften ... 1284
23.7.3 Fill-Methode ... 1285
23.7.4 Update-Methode ... 1286
23.8 Praxisbeispiele ... 1287
23.8.1 Wichtige ADO.NET-Objekte im Einsatz ... 1287
23.8.2 Eine Aktionsabfrage ausführen ... 1288
23.8.3 Eine Auswahlabfrage aufrufen ... 1291
23.8.4 Die Datenbank aktualisieren ... 1293

24 Das DataSet ... 1297

24.1 Grundlegende Features des DataSets ... 1297
24.1.1 Die Objekthierarchie ... 1298
24.1.2 Die wichtigsten Klassen ... 1298
24.1.3 Erzeugen eines DataSets ... 1299
24.2 Das DataTable-Objekt ... 1300
24.2.1 DataTable erzeugen ... 1301
24.2.2 Spalten hinzufügen ... 1301
24.2.3 Zeilen zur DataTable hinzufügen ... 1302
24.2.4 Auf den Inhalt einer DataTable zugreifen ... 1303
24.3 Die DataView ... 1305
24.3.1 Erzeugen einer DataView ... 1305
24.3.2 Sortieren und Filtern von Datensätzen ... 1305
24.3.3 Suchen von Datensätzen ... 1306
24.4 Typisierte DataSets ... 1306
24.4.1 Ein typisiertes DataSet erzeugen ... 1307
24.4.2 Das Konzept der Datenquellen ... 1308
24.4.3 Typisierte DataSets und TableAdapter ... 1309

24.5 Die Qual der Wahl ... 1310
24.5.1 DataReader – der schnelle Lesezugriff ... 1311
24.5.2 DataSet – die Datenbank im Hauptspeicher ... 1311
24.5.3 Objektrelationales Mapping – die Zukunft? ... 1312
24.6 Praxisbeispiele ... 1313
24.6.1 In der DataView sortieren und filtern ... 1313
24.6.2 Suche nach Datensätzen ... 1315
24.6.3 Ein DataSet in einen XML-String serialisieren ... 1316
24.6.4 Untypisiertes in ein typisiertes DataSet konvertieren ... 1321
24.6.5 Eine LINQ to SQL-Abfrage ausführen ... 1326

25 OOP-Spezial ... 1331

25.1 Eine kleine Einführung in die UML ... 1331
25.1.1 Use Case-Diagramm ... 1331
25.1.2 Use Case-Dokumentation ... 1333
25.1.3 Objekte identifizieren ... 1334
25.1.4 Statisches Modell ... 1335
25.1.5 Beziehungen zwischen den Klassen ... 1336
25.1.6 Dynamisches Modell ... 1336
25.1.7 Implementierung ... 1337
25.1.8 Test-Client ... 1341
25.2 Der Klassen-Designer ... 1344
25.2.1 Ein neues Klassendiagramm erzeugen ... 1345
25.2.2 Werkzeugkasten ... 1346
25.2.3 Enumeration ... 1347
25.2.4 Klasse ... 1349
25.2.5 Struktur ... 1351
25.2.6 Abstrakte Klasse ... 1351
25.2.7 Schnittstelle ... 1353
25.2.8 Delegate ... 1355
25.2.9 Zuordnung ... 1357
25.2.10 Vererbung ... 1358
25.2.11 Diagramme anpassen ... 1358
25.2.12 Wann lohnt sich der Klassen-Designer? ... 1359
25.3 Praxisbeispiele ... 1359
25.3.1 Implementierung einer Finite State Machine ... 1359
25.3.2 Modellierung des Bestellsystems einer Firma ... 1365

26 Das Microsoft Event Pattern ... 1379

26.1 Einführung in Design Pattern ... 1379
26.2 Aufbau und Bedeutung des Observer Pattern ... 1380
26.2.1 Subjekt und Observer ... 1380
26.2.2 Sequenzdiagramme ... 1382
26.2.3 Die Registration-Sequenz ... 1382
26.2.4 Die Notification-Sequenz ... 1383
26.2.5 Die Unregistration-Sequenz ... 1383
26.2.6 Bedeutung der Sequenzen für das Geschäftsmodell ... 1384
26.2.7 Die Rolle des Containers ... 1384
26.3 Implementieren mit Interfaces und Callbacks ... 1385
26.3.1 Übersicht und Klassendiagramm ... 1385
26.3.2 Die Schnittstellen IObserver und IObservable ... 1387
26.3.3 Die Abstrakte Klasse Subject ... 1387
26.3.4 Observer1 ... 1388
26.3.5 Observer2 ... 1389
26.3.6 Model ... 1390
26.3.7 Form1 ... 1391
26.3.8 Ein zweites Klassendiagramm ... 1392
26.3.9 Testen ... 1393
26.4 Implementierung mit Delegates und Events ... 1394
26.4.1 Multicast-Events ... 1395
26.4.2 IObserver, IObservable und Subject ... 1395
26.4.3 Observer1 und Observer2 ... 1396
26.4.4 Model ... 1396
26.4.5 Form1 ... 1397
26.4.6 Test und Vergleich ... 1397
26.4.7 Klassendiagramm ... 1398
26.5 Implementierung des Microsoft Event Pattern ... 1399
26.5.1 Namensgebung für Ereignisse ... 1399
26.5.2 Namensgebung und Signatur der Delegates ... 1399
26.5.3 Hinzufügen einer das Ereignis auslösenden Methode ... 1400
26.5.4 Model ... 1401
26.5.5 Observer1 ... 1402
26.5.6 Form1 ... 1403
26.6 Test und Vergleich ... 1403
26.7 Klassendiagramm ... 1403

26.8 Implementierung eines Event Pattern ... 1404
26.9 Praxisbeispiel ... 1406
26.9.1 Objekte beobachten sich gegenseitig ... 1406

27 Verteilen von Anwendungen ... 1417

27.1 ClickOnce-Deployment ... 1418
27.1.1 Übersicht/Einschränkungen ... 1418
27.1.2 Die Vorgehensweise ... 1419
27.1.3 Ort der Veröffentlichung ... 1419
27.1.4 Anwendungsdateien ... 1420
27.1.5 Erforderliche Komponenten ... 1420
27.1.6 Aktualisierungen ... 1421
27.1.7 Veröffentlichungsoptionen ... 1422
27.1.8 Veröffentlichen ... 1423
27.1.9 Verzeichnisstruktur ... 1423
27.1.10 Der Webpublishing-Assistent ... 1425
27.1.11 Neue Versionen erstellen ... 1426
27.2 InstallShield ... 1426
27.2.1 Installation ... 1426
27.2.2 Aktivieren ... 1427
27.2.3 Ein neues Setup-Projekt ... 1427
27.2.4 Finaler Test ... 1435

28 Weitere Techniken ... 1437

28.1 Zugriff auf die Zwischenablage ... 1437
28.1.1 Das Clipboard-Objekt ... 1437
28.1.2 Zwischenablage-Funktionen für Textboxen ... 1439
28.2 Arbeiten mit der Registry ... 1439
28.2.1 Allgemeines ... 1440
28.2.2 Registry-Unterstützung in .NET ... 1442
28.3 .NET-Reflection ... 1443
28.3.1 Übersicht ... 1443
28.3.2 Assembly laden ... 1443
28.3.3 Mittels GetType und Type Informationen sammeln ... 1444
28.3.4 Dynamisches Laden von Assemblies ... 1446
28.4 Das SerialPort-Control ... 1449
28.4.1 Übersicht ... 1449
28.4.2 Einführungsbeispiele ... 1450

28.4.3 Thread-Probleme bei Windows Forms-Anwendungen ... 1453
28.4.4 Ein einfaches Terminalprogramm ... 1456
28.5 Praxisbeispiele ... 1461
28.5.1 Zugriff auf die Registry ... 1461
28.5.2 Dateiverknüpfungen erzeugen ... 1463
28.5.3 Betrachter für Manifestressourcen ... 1465
28.5.4 Ressourcen mit Reflection auslesen ... 1468

29 Konsolenanwendungen ... 1471

29.1 Grundaufbau/Konzepte ... 1471
29.1.1 Unser Hauptprogramm – Program.cs ... 1472
29.1.2 Rückgabe eines Fehlerstatus ... 1473
29.1.3 Parameterübergabe ... 1474
29.1.4 Zugriff auf die Umgebungsvariablen ... 1475
29.2 Die Kommandozentrale: System.Console ... 1476
29.2.1 Eigenschaften ... 1477
29.2.2 Methoden/Ereignisse ... 1477
29.2.3 Textausgaben ... 1478
29.2.4 Farbangaben ... 1479
29.2.5 Tastaturabfragen ... 1480
29.2.6 Arbeiten mit Streamdaten ... 1481
29.3 Praxisbeispiel ... 1483
29.3.1 Farbige Konsolenanwendung ... 1483
29.3.2 Weitere Hinweise und Beispiele ... 1485

Teil VI: Windows Forms

30 Windows Forms-Anwendungen ... 1489

30.1 Grundaufbau/Konzepte ... 1489
30.1.1 Das Hauptprogramm – Program.cs ... 1490
30.1.2 Die Oberflächendefinition – Form1.Designer.cs ... 1494
30.1.3 Die Spielwiese des Programmierers – Form1.cs ... 1495
30.1.4 Die Datei AssemblyInfo.cs ... 1496
30.1.5 Resources resx/Resources.Designer.cs ... 1497
30.1.6 Settings.settings/Settings.Designer.cs ... 1498
30.1.7 Settings.cs ... 1499

30.2 Ein Blick auf die Application-Klasse ... 1500
30.2.1 Eigenschaften ... 1500
30.2.2 Methoden ... 1501
30.2.3 Ereignisse ... 1503
30.3 Allgemeine Eigenschaften von Komponenten ... 1503
30.3.1 Font ... 1504
30.3.2 Handle ... 1506
30.3.3 Tag ... 1507
30.3.4 Modifiers ... 1507
30.4 Allgemeine Ereignisse von Komponenten ... 1508
30.4.1 Die Eventhandler-Argumente ... 1508
30.4.2 Sender ... 1508
30.4.3 Der Parameter e ... 1510
30.4.4 Mausereignisse ... 1510
30.4.5 KeyPreview ... 1512
30.4.6 Weitere Ereignisse ... 1513
30.4.7 Validätsprüfungen ... 1513
30.4.8 SendKeys ... 1514
30.5 Allgemeine Methoden von Komponenten ... 1515

31 Windows Forms-Formulare ... 1517

31.1 Übersicht ... 1517
31.1.1 Wichtige Eigenschaften des Form-Objekts ... 1518
31.1.2 Wichtige Ereignisse des Form-Objekts ... 1520
31.1.3 Wichtige Methoden des Form-Objekts ... 1521
31.2 Praktische Aufgabenstellungen ... 1522
31.2.1 Fenster anzeigen ... 1522
31.2.2 Splash Screens beim Anwendungsstart anzeigen ... 1525
31.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen ... 1528
31.2.4 Ein Formular durchsichtig machen ... 1528
31.2.5 Die Tabulatorreihenfolge festlegen ... 1529
31.2.6 Ausrichten von Komponenten im Formular ... 1530
31.2.7 Spezielle Panels für flexible Layouts ... 1532
31.2.8 Menüs erzeugen ... 1534
31.3 MDI-Anwendungen ... 1538
31.3.1 "Falsche" MDI-Fenster bzw. Verwenden von Parent ... 1538
31.3.2 Die echten MDI-Fenster ... 1539
31.3.3 Die Kindfenster ... 1540

31.3.4 Automatisches Anordnen der Kindfenster ... 1541
31.3.5 Zugriff auf die geöffneten MDI-Kindfenster ... 1542
31.3.6 Zugriff auf das aktive MDI-Kindfenster ... 1543
31.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü ... 1543
31.4 Praxisbeispiele ... 1544
31.4.1 Informationsaustausch zwischen Formularen ... 1544
31.4.2 Ereigniskette beim Laden/Entladen eines Formulars ... 1551

32 Windows Forms-Komponenten ... 1557

32.1 Allgemeine Hinweise ... 1557
32.1.1 Hinzufügen von Komponenten ... 1557
32.1.2 Komponenten zur Laufzeit per Code erzeugen ... 1558
32.2 Allgemeine Steuerelemente ... 1560
32.2.1 Label ... 1560
32.2.2 LinkLabel ... 1561
32.2.3 Button ... 1562
32.2.4 TextBox ... 1563
32.2.5 MaskedTextBox ... 1566
32.2.6 CheckBox ... 1567
32.2.7 RadioButton ... 1569
32.2.8 ListBox ... 1569
32.2.9 CheckedListBox ... 1571
32.2.10 ComboBox ... 1571
32.2.11 PictureBox ... 1572
32.2.12 DateTimePicker ... 1573
32.2.13 MonthCalendar ... 1573
32.2.14 HScrollBar, VScrollBar ... 1574
32.2.15 TrackBar ... 1575
32.2.16 NumericUpDown ... 1575
32.2.17 DomainUpDown ... 1576
32.2.18 ProgressBar ... 1576
32.2.19 RichTextBox ... 1577
32.2.20 ListView ... 1578
32.2.21 TreeView ... 1584
32.2.22 WebBrowser ... 1589
32.3 Container ... 1590
32.3.1 FlowLayout/TableLayout/SplitContainer ... 1590
32.3.2 Panel ... 1590

32.3.3 GroupBox ... 1591
32.3.4 TabControl ... 1592
32.3.5 ImageList ... 1594
32.4 Menüs & Symbolleisten ... 1595
32.4.1 MenuStrip und ContextMenuStrip ... 1595
32.4.2 ToolStrip ... 1595
32.4.3 StatusStrip ... 1595
32.4.4 ToolStripContainer ... 1596
32.5 Daten ... 1596
32.5.1 DataSet ... 1596
32.5.2 DataGridView/DataGrid ... 1597
32.5.3 BindingNavigator/BindingSource ... 1597
32.5.4 Chart ... 1597
32.6 Komponenten ... 1599
32.6.1 ErrorProvider ... 1599
32.6.2 HelpProvider ... 1599
32.6.3 ToolTip ... 1599
32.6.4 BackgroundWorker ... 1599
32.6.5 Timer ... 1599
32.6.6 SerialPort ... 1600
32.7 Drucken ... 1600
32.7.1 PrintPreviewControl ... 1600
32.7.2 PrintDocument ... 1600
32.8 Dialoge ... 1600
32.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog ... 1600
32.8.2 FontDialog/ColorDialog ... 1601
32.9 WPF-Unterstützung mit dem ElementHost ... 1601
32.10 Praxisbeispiele ... 1601
32.10.1 Mit der CheckBox arbeiten ... 1601
32.10.2 Steuerelemente per Code selbst erzeugen ... 1603
32.10.3 Controls-Auflistung im TreeView anzeigen ... 1605
32.10.4 WPF-Komponenten mit dem ElementHost anzeigen ... 1609

33 Grundlagen Grafikausgabe ... 1613

33.1 Übersicht und erste Schritte ... 1613
33.1.1 GDI+ – Ein erster Einblick für Umsteiger ... 1614
33.1.2 Namespaces für die Grafikausgabe ... 1615

33.2 Darstellen von Grafiken ... 1617
33.2.1 Die PictureBox-Komponente ... 1617
33.2.2 Das Image-Objekt ... 1618
33.2.3 Laden von Grafiken zur Laufzeit ... 1619
33.2.4 Sichern von Grafiken ... 1619
33.2.5 Grafikeigenschaften ermitteln ... 1620
33.2.6 Erzeugen von Vorschaugrafiken (Thumbnails) ... 1621
33.2.7 Die Methode RotateFlip ... 1622
33.2.8 Skalieren von Grafiken ... 1623
33.3 Das .NET-Koordinatensystem ... 1624
33.3.1 Globale Koordinaten ... 1625
33.3.2 Seitenkoordinaten (globale Transformation) ... 1626
33.3.3 Gerätekoordinaten (Seitentransformation) ... 1628
33.4 Grundlegende Zeichenfunktionen von GDI+ ... 1629
33.4.1 Das zentrale Graphics-Objekt ... 1629
33.4.2 Punkte zeichnen/abfragen ... 1632
33.4.3 Linien ... 1633
33.4.4 Kantenglättung mit Antialiasing ... 1634
33.4.5 PolyLine ... 1635
33.4.6 Rechtecke ... 1635
33.4.7 Polygone ... 1637
33.4.8 Splines ... 1638
33.4.9 Bézierkurven ... 1639
33.4.10 Kreise und Ellipsen ... 1640
33.4.11 Tortenstück (Segment) ... 1640
33.4.12 Bogenstück ... 1642
33.4.13 Wo sind die Rechtecke mit den runden Ecken? ... 1643
33.4.14 Textausgabe ... 1644
33.4.15 Ausgabe von Grafiken ... 1648
33.5 Unser Werkzeugkasten ... 1649
33.5.1 Einfache Objekte ... 1649
33.5.2 Vordefinierte Objekte ... 1651
33.5.3 Farben/Transparenz ... 1653
33.5.4 Stifte (Pen) ... 1654
33.5.5 Pinsel (Brush) ... 1657
33.5.6 SolidBrush ... 1658
33.5.7 HatchBrush ... 1658
33.5.8 TextureBrush ... 1659

33.5.9 LinearGradientBrush ... 1660
33.5.10 PathGradientBrush ... 1661
33.5.11 Fonts ... 1662
33.5.12 Path-Objekt ... 1663
33.5.13 Clipping/Region ... 1666
33.6 Standarddialoge ... 1670
33.6.1 Schriftauswahl ... 1670
33.6.2 Farbauswahl ... 1671
33.7 Praxisbeispiele ... 1673
33.7.1 Ein Graphics-Objekt erzeugen ... 1673
33.7.2 Zeichenoperationen mit der Maus realisieren ... 1675

34 Druckausgabe ... 1679

34.1 Einstieg und Übersicht ... 1679
34.1.1 Nichts geht über ein Beispiel ... 1679
34.1.2 Programmiermodell ... 1681
34.1.3 Kurzübersicht der Objekte ... 1682
34.2 Auswerten der Druckereinstellungen ... 1682
34.2.1 Die vorhandenen Drucker ... 1682
34.2.2 Der Standarddrucker ... 1683
34.2.3 Verfügbare Papierformate/Seitenabmessungen ... 1684
34.2.4 Der eigentliche Druckbereich ... 1685
34.2.5 Die Seitenausrichtung ermitteln ... 1686
34.2.6 Ermitteln der Farbfähigkeit ... 1686
34.2.7 Die Druckauflösung abfragen ... 1686
34.2.8 Ist beidseitiger Druck möglich? ... 1687
34.2.9 Einen "Informationsgerätekontext" erzeugen ... 1687
34.2.10 Abfragen von Werten während des Drucks ... 1688
34.3 Festlegen von Druckereinstellungen ... 1689
34.3.1 Einen Drucker auswählen ... 1689
34.3.2 Drucken in Millimetern ... 1689
34.3.3 Festlegen der Seitenränder ... 1690
34.3.4 Druckjobname ... 1691
34.3.5 Anzahl der Kopien ... 1692
34.3.6 Beidseitiger Druck ... 1692
34.3.7 Seitenzahlen festlegen ... 1693
34.3.8 Druckqualität verändern ... 1696
34.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen ... 1697

34.4 Die Druckdialoge verwenden ... 1697
34.4.1 PrintDialog ... 1698
34.4.2 PageSetupDialog ... 1699
34.4.3 PrintPreviewDialog ... 1701
34.4.4 Ein eigenes Druckvorschau-Fenster realisieren ... 1702
34.5 Drucken mit OLE-Automation ... 1703
34.5.1 Kurzeinstieg in die OLE-Automation ... 1703
34.5.2 Drucken mit Microsoft Word ... 1705
34.6 Praxisbeispiele ... 1707
34.6.1 Den Drucker umfassend konfigurieren ... 1707
34.6.2 Diagramme mit dem Chart-Control drucken ... 1717
34.6.3 Druckausgabe mit Word ... 1719

35 Windows Forms-Datenbindung ... 1725

35.1 Prinzipielle Möglichkeiten ... 1725
35.2 Manuelle Bindung an einfache Datenfelder ... 1726
35.2.1 BindingSource erzeugen ... 1726
35.2.2 Binding-Objekt ... 1727
35.2.3 DataBindings-Collection ... 1727
35.3 Manuelle Bindung an Listen und Tabellen ... 1727
35.3.1 DataGridView ... 1728
35.3.2 Datenbindung von ComboBox und ListBox ... 1728
35.4 Entwurfszeit-Bindung an typisierte DataSets ... 1728
35.5 Drag & Drop-Datenbindung ... 1730
35.6 Navigations- und Bearbeitungsfunktionen ... 1730
35.6.1 Navigieren zwischen den Datensätzen ... 1730
35.6.2 Hinzufügen und Löschen ... 1730
35.6.3 Aktualisieren und Abbrechen ... 1731
35.6.4 Verwendung des BindingNavigators ... 1731
35.7 Die Anzeigedaten formatieren ... 1732
35.8 Praxisbeispiele ... 1732
35.8.1 Einrichten und Verwenden einer Datenquelle ... 1732
35.8.2 Eine Auswahlabfrage im DataGridView anzeigen ... 1736
35.8.3 Master-Detailbeziehungen im DataGrid anzeigen ... 1739
35.8.4 Datenbindung Chart-Control ... 1740

36 Erweiterte Grafikausgabe 1745

36.1 Transformieren mit der Matrix-Klasse 1745
36.1.1 Übersicht 1745
36.1.2 Translation 1746
36.1.3 Skalierung 1746
36.1.4 Rotation 1747
36.1.5 Scherung 1747
36.1.6 Zuweisen der Matrix 1748
36.2 Low-Level-Grafikmanipulationen 1748
36.2.1 Worauf zeigt Scan0? 1749
36.2.2 Anzahl der Spalten bestimmen 1750
36.2.3 Anzahl der Zeilen bestimmen 1751
36.2.4 Zugriff im Detail (erster Versuch) 1751
36.2.5 Zugriff im Detail (zweiter Versuch) 1753
36.2.6 Invertieren 1755
36.2.7 In Graustufen umwandeln 1756
36.2.8 Heller/Dunkler 1757
36.2.9 Kontrast 1759
36.2.10 Gamma-Wert 1760
36.2.11 Histogramm spreizen 1760
36.2.12 Ein universeller Grafikfilter 1763
36.3 Fortgeschrittene Techniken 1767
36.3.1 Flackerfrei dank Double Buffering 1767
36.3.2 Animationen 1769
36.3.3 Animated GIFs 1772
36.3.4 Auf einzelne GIF-Frames zugreifen 1774
36.3.5 Transparenz realisieren 1776
36.3.6 Eine Grafik maskieren 1777
36.3.7 JPEG-Qualität beim Sichern bestimmen 1779
36.4 Grundlagen der 3D-Vektorgrafik 1780
36.4.1 Datentypen für die Verwaltung 1780
36.4.2 Eine universelle 3D-Grafik-Klasse 1781
36.4.3 Grundlegende Betrachtungen 1782
36.4.4 Translation 1785
36.4.5 Streckung/Skalierung 1786
36.4.6 Rotation 1787
36.4.7 Die eigentlichen Zeichenroutinen 1789

36.5 Und doch wieder GDI-Funktionen ... 1791
36.5.1 Am Anfang war das Handle ... 1791
36.5.2 Gerätekontext (Device Context Types) ... 1794
36.5.3 Koordinatensysteme und Abbildungsmodi ... 1796
36.5.4 Zeichenwerkzeuge/Objekte ... 1800
36.5.5 Bitmaps ... 1802
36.6 Praxisbeispiele ... 1806
36.6.1 Die Transformationsmatrix verstehen ... 1806
36.6.2 Eine 3D-Grafikausgabe in Aktion ... 1809
36.6.3 Einen Fenster-Screenshot erzeugen ... 1812

37 Ressourcen/Lokalisierung ... 1815

37.1 Manifestressourcen ... 1815
37.1.1 Erstellen von Manifestressourcen ... 1815
37.1.2 Zugriff auf Manifestressourcen ... 1817
37.2 Typisierte Ressourcen ... 1819
37.2.1 Erzeugen von resources-Dateien ... 1819
37.2.2 Hinzufügen der resources-Datei zum Projekt ... 1819
37.2.3 Zugriff auf die Inhalte von resources-Dateien ... 1820
37.2.4 ResourceManager einer resources-Datei erzeugen ... 1820
37.2.5 Was sind resx-Dateien? ... 1821
37.3 Streng typisierte Ressourcen ... 1821
37.3.1 Erzeugen streng typisierter Ressourcen ... 1822
37.3.2 Verwenden streng typisierter Ressourcen ... 1822
37.3.3 Streng typisierte Ressourcen per Reflection auslesen ... 1823
37.4 Anwendungen lokalisieren ... 1825
37.4.1 Localizable und Language ... 1825
37.4.2 Beispiel "Landesfahnen" ... 1825
37.4.3 Einstellen der aktuellen Kultur zur Laufzeit ... 1828

38 Komponentenentwicklung ... 1831

38.1 Überblick ... 1831
38.2 Benutzersteuerelement ... 1832
38.2.1 Entwickeln einer Auswahl-ListBox ... 1832
38.2.2 Komponente verwenden ... 1834
38.3 Benutzerdefiniertes Steuerelement ... 1835
38.3.1 Entwickeln eines BlinkLabels ... 1835
38.3.2 Verwenden der Komponente ... 1838

38.4 Komponentenklasse ... 1838
38.5 Eigenschaften ... 1839
38.5.1 Einfache Eigenschaften ... 1839
38.5.2 Schreib-/Lesezugriff (Get/Set) ... 1839
38.5.3 Nur Lese-Eigenschaft (ReadOnly) ... 1840
38.5.4 Nur-Schreibzugriff (WriteOnly) ... 1841
38.5.5 Hinzufügen von Beschreibungen ... 1841
38.5.6 Ausblenden im Eigenschaftenfenster ... 1841
38.5.7 Einfügen in Kategorien ... 1842
38.5.8 Default-Wert einstellen ... 1842
38.5.9 Standard-Eigenschaft (Indexer) ... 1843
38.5.10 Wertebereichsbeschränkung und Fehlerprüfung ... 1843
38.5.11 Eigenschaften von Aufzählungstypen ... 1845
38.5.12 Standard Objekt-Eigenschaften ... 1846
38.5.13 Eigene Objekt-Eigenschaften ... 1846
38.6 Methoden ... 1848
38.6.1 Konstruktor ... 1849
38.6.2 Class-Konstruktor ... 1850
38.6.3 Destruktor ... 1851
38.6.4 Aufruf des Basisklassen-Konstruktors ... 1851
38.6.5 Aufruf von Basisklassen-Methoden ... 1852
38.7 Ereignisse (Events) ... 1852
38.7.1 Ereignis mit Standardargument definieren ... 1853
38.7.2 Ereignis mit eigenen Argumenten ... 1854
38.7.3 Ein Default-Ereignis festlegen ... 1855
38.7.4 Mit Ereignissen auf Windows-Messages reagieren ... 1855
38.8 Weitere Themen ... 1857
38.8.1 Wohin mit der Komponente? ... 1857
38.8.2 Assembly-Informationen festlegen ... 1858
38.8.3 Assemblies signieren ... 1861
38.8.4 Komponenten-Ressourcen einbetten ... 1861
38.8.5 Der Komponente ein Icon zuordnen ... 1862
38.8.6 Den Designmodus erkennen ... 1863
38.8.7 Komponenten lizenzieren ... 1863
38.9 Praxisbeispiele ... 1867
38.9.1 AnimGif – Anzeige von Animationen ... 1867
38.9.2 Eine FontComboBox entwickeln ... 1870
38.9.3 Das PropertyGrid verwenden ... 1872

Teil VII: ASP.NET

39 Einführung in ASP.NET ... 1877

39.1 ASP.NET für Ein- und Umsteiger ... 1877
39.1.1 ASP – Ein kurzer Blick zurück ... 1877
39.1.2 Was ist bei ASP.NET anders? ... 1878
39.1.3 Was gibt es noch in ASP.NET? ... 1880
39.1.4 Vorteile von ASP.NET gegenüber ASP ... 1881
39.1.5 Voraussetzungen für den Einsatz von ASP.NET ... 1882
39.1.6 Und was hat das alles mit C# zu tun? ... 1882
39.2 Unsere erste Web-Anwendung ... 1885
39.2.1 Visueller Entwurf der Bedienoberfläche ... 1885
39.2.2 Zuweisen der Objekteigenschaften ... 1888
39.2.3 Verknüpfen der Objekte mit Ereignissen ... 1889
39.2.4 Programm kompilieren und testen ... 1890
39.3 Die ASP.NET-Projektdateien ... 1891
39.3.1 Die ASP.NET-Projekttypen ... 1892
39.3.2 ASPX-Datei(en) ... 1894
39.3.3 Die aspx.cs-Datei(en) ... 1896
39.3.4 Die Datei Global.asax ... 1897
39.3.5 Das Startformular ... 1898
39.3.6 Die Datei Web.config ... 1898
39.3.7 Masterpages (master-Dateien) ... 1901
39.3.8 Sitemap (Web.sitemap) ... 1901
39.3.9 Benutzersteuerelemente (ascx-Dateien) ... 1902
39.3.10 Die Web-Projekt-Verzeichnisse ... 1902
39.4 Lernen am Beispiel ... 1903
39.4.1 Erstellen des Projekts ... 1903
39.4.2 Oberflächengestaltung ... 1904
39.4.3 Ereignisprogrammierung ... 1905
39.4.4 Ein Fehler, was nun? ... 1907
39.4.5 Ereignisse von Textboxen ... 1908
39.4.6 Ein gemeinsamer Ereignis-Handler ... 1909
39.4.7 Eingabefokus setzen ... 1909
39.4.8 Ausgaben in einer Tabelle ... 1910
39.4.9 Scrollen der Anzeige ... 1912
39.4.10 Zusammenspiel mehrerer Formulare ... 1913
39.4.11 Umleiten bei Direktaufruf ... 1914

39.4.12 Ärger mit den Cookies ... 1915
39.4.13 Export auf den IIS ... 1917
39.5 Tipps & Tricks ... 1918
39.5.1 Nachinstallieren IIS 7 bzw. 7.5 (Windows 7) ... 1918
39.5.2 Nachinstallieren IIS8 (Windows 8) ... 1919

40 Übersicht ASP.NET-Controls ... 1921

40.1 Einfache Steuerelemente im Überblick ... 1921
40.1.1 Label ... 1921
40.1.2 TextBox ... 1923
40.1.3 Button, ImageButton, LinkButton ... 1924
40.1.4 CheckBox, RadioButton ... 1925
40.1.5 CheckBoxList, BulletList, RadioButtonList ... 1926
40.1.6 Table ... 1927
40.1.7 Hyperlink ... 1929
40.1.8 Image, ImageMap ... 1929
40.1.9 Calendar ... 1931
40.1.10 Panel ... 1932
40.1.11 HiddenField ... 1932
40.1.12 Substitution ... 1933
40.1.13 XML ... 1934
40.1.14 FileUpload ... 1936
40.1.15 AdRotator ... 1937
40.2 Steuerelemente für die Seitennavigation ... 1938
40.2.1 Mehr Übersicht mit Web.Sitemap ... 1938
40.2.2 Menu ... 1940
40.2.3 TreeView ... 1943
40.2.4 SiteMapPath ... 1946
40.2.5 MultiView, View ... 1947
40.2.6 Wizard ... 1948
40.3 Webseitenlayout/-design ... 1950
40.3.1 Masterpages ... 1950
40.3.2 Themes/Skins ... 1953
40.3.3 Webparts ... 1956
40.4 Die Validator-Controls ... 1957
40.4.1 Übersicht ... 1957
40.4.2 Wo findet die Fehlerprüfung statt? ... 1958
40.4.3 Verwendung ... 1958

40.4.4 RequiredFieldValidator ... 1959
40.4.5 CompareValidator ... 1960
40.4.6 RangeValidator ... 1962
40.4.7 RegularExpressionValidator ... 1962
40.4.8 CustomValidator ... 1963
40.4.9 ValidationSummary ... 1966
40.4.10 Weitere Möglichkeiten ... 1967
40.5 Praxisbeispiele ... 1967
40.5.1 Themes und Skins verstehen ... 1967
40.5.2 Masterpages verwenden ... 1972
40.5.3 Webparts verwenden ... 1975

41 Datenbindung unter ASP.NET ... 1981

41.1 Einstiegsbeispiel ... 1981
41.1.1 Erstellen der ASP.NET-Website ... 1981
41.2 Einführung ... 1986
41.2.1 Konzept ... 1986
41.2.2 Übersicht über die DataSource-Steuerelemente ... 1987
41.3 SQLDataSource ... 1988
41.3.1 Datenauswahl mit Parametern ... 1990
41.3.2 Parameter für INSERT, UPDATE und DELETE ... 1991
41.3.3 Methoden ... 1993
41.3.4 Caching ... 1994
41.3.5 Aktualisieren/Refresh ... 1995
41.4 AccessDataSource ... 1995
41.5 ObjectDataSource ... 1995
41.5.1 Verbindung zwischen Objekt und DataSource ... 1995
41.5.2 Ein Beispiel sorgt für Klarheit ... 1997
41.5.3 Geschäftsobjekte in einer Session verwalten ... 2001
41.6 SitemapDataSource ... 2003
41.7 LinqDataSource ... 2004
41.7.1 Bindung von einfachen Collections ... 2004
41.7.2 Bindung eines LINQ to SQL-DataContext ... 2005
41.8 EntityDataSource ... 2007
41.8.1 Entity Data Model erstellen ... 2007
41.8.2 EntityDataSource anbinden ... 2010
41.8.3 Datenmenge filtern ... 2013
41.9 XmlDataSource ... 2013

41.10 QueryExtender ... 2015
41.10.1 Grundlagen ... 2015
41.10.2 Suchen ... 2016
41.10.3 Sortieren ... 2018
41.11 GridView ... 2019
41.11.1 Auswahlfunktion (Zeilenauswahl) ... 2019
41.11.2 Auswahl mit mehrspaltigem Index ... 2020
41.11.3 Hyperlink-Spalte für Detailansicht ... 2020
41.11.4 Spalten erzeugen ... 2021
41.11.5 Paging realisieren ... 2022
41.11.6 Edit, Update, Delete ... 2024
41.11.7 Keine Daten, was tun? ... 2024
41.12 DetailsView ... 2024
41.13 FormView ... 2026
41.14 DataList ... 2029
41.14.1 Bearbeitungsfunktionen implementieren ... 2030
41.14.2 Layout verändern ... 2031
41.15 Repeater ... 2032
41.16 ListView ... 2033
41.17 Typisierte Datenbindung ... 2033
41.18 Model Binding ... 2034
41.19 Chart ... 2036

42 ASP.NET-Objekte/-Techniken ... 2039

42.1 Wichtige ASP.NET-Objekte ... 2039
42.1.1 HTTPApplication ... 2039
42.1.2 Application ... 2042
42.1.3 Session ... 2043
42.1.4 Page ... 2045
42.1.5 Request ... 2048
42.1.6 Response ... 2051
42.1.7 Server ... 2055
42.1.8 Cookies verwenden ... 2056
42.2 Fehlerbehandlung unter ASP.NET ... 2059
42.2.1 Fehler beim Entwurf ... 2059
42.2.2 Laufzeitfehler ... 2059
42.2.3 Eine eigene Fehlerseite ... 2061
42.2.4 Fehlerbehandlung im Web Form ... 2062

42.2.5 Fehlerbehandlung in der Anwendung ... 2063
42.2.6 Alternative Fehlerseite einblenden ... 2064
42.2.7 Lokale Fehlerbehandlung ... 2065
42.2.8 Seite nicht gefunden – was nun? ... 2066
42.3 E-Mail-Versand in ASP.NET ... 2066
42.3.1 Übersicht ... 2067
42.3.2 Mail-Server bestimmen ... 2067
42.3.3 Einfache Text-E-Mails versenden ... 2069
42.3.4 E-Mails mit Dateianhang ... 2070
42.4 Sicherheit von Webanwendungen ... 2071
42.4.1 Authentication ... 2071
42.4.2 Forms Authentication realisieren ... 2072
42.4.3 Impersonation ... 2076
42.4.4 Authorization ... 2077
42.4.5 Administrieren der Website ... 2079
42.4.6 Steuerelemente für das Login-Handling ... 2083
42.4.7 Programmieren der Sicherheitseinstellungen ... 2087
42.5 AJAX in ASP.NET-Anwendungen ... 2089
42.5.1 Was ist AJAX und was kann es? ... 2089
42.5.2 Die AJAX-Controls ... 2090
42.5.3 AJAX-Control-Toolkit ... 2094
42.6 User Controls/Webbenutzersteuerelemente ... 2095
42.6.1 Ein simples Einstiegsbeispiel ... 2096
42.6.2 Dynamische Grafiken im User Control anzeigen ... 2099
42.6.3 Grafikausgaben per User Control realisieren ... 2104

Teil VIII: Silverlight

43 Silverlight-Entwicklung ... 2109
43.1 Einführung ... 2109
43.1.1 Zielplattformen ... 2110
43.1.2 Silverlight-Applikationstypen ... 2110
43.1.3 Wichtige Unterschiede zu den WPF-Anwendungen ... 2112
43.1.4 Vor- und Nachteile von Silverlight-Anwendungen ... 2114
43.1.5 Entwicklungstools ... 2116
43.1.6 Installation auf dem Client ... 2116

43.2 Die Silverlight-Anwendung im Detail ... 2117
43.2.1 Ein kleines Beispielprojekt ... 2118
43.2.2 Das Application Package und das Test-Web ... 2120
43.3 Die Projektdateien im Überblick ... 2123
43.3.1 Projektverwaltung mit App.xaml & App.xaml.cs ... 2124
43.3.2 MainPage.xaml & MainPage.xaml.cs ... 2126
43.3.3 AssemblyInfo.cs ... 2127
43.4 Fenster und Seiten in Silverlight ... 2127
43.4.1 Das Standardfenster ... 2128
43.4.2 Untergeordnete Silverlight-Fenster ... 2129
43.4.3 UserControls für die Anzeige von Detaildaten ... 2131
43.4.4 Echte Windows ... 2132
43.4.5 Navigieren in Silverlight-Anwendungen ... 2133
43.5 Datenbanken/Datenbindung ... 2138
43.5.1 ASP.NET-Webdienste/WCF-Dienste ... 2139
43.5.2 WCF Data Services ... 2148
43.6 Isolierter Speicher ... 2159
43.6.1 Grundkonzept ... 2159
43.6.2 Das virtuelle Dateisystem verwalten ... 2160
43.6.3 Arbeiten mit Dateien ... 2163
43.7 Fulltrust-Anwendungen ... 2164
43.8 Praxisbeispiele ... 2167
43.8.1 Eine Out-of-Browser-Applikation realisieren ... 2167
43.8.2 Out-of-Browser-Anwendung aktualisieren ... 2171
43.8.3 Testen auf aktive Internetverbindung ... 2172
43.8.4 Auf Out-of-Browser-Anwendung testen ... 2173
43.8.5 Den Browser bestimmen ... 2173
43.8.6 Parameter an das Plug-in übergeben ... 2174
43.8.7 Auf den QueryString zugreifen ... 2176
43.8.8 Timer in Silverlight nutzen ... 2177
43.8.9 Dateien lokal speichern ... 2178
43.8.10 Drag & Drop realisieren ... 2180
43.8.11 Auf die Zwischenablage zugreifen ... 2181
43.8.12 Weitere Fenster öffnen ... 2183

Index ... 2187

Zweites Vorwort

Leider hat die Fülle des Materials und das Einsatzspektrum von Visual Studio mittlerweile alle Grenzen gesprengt (Sie sehen es sicher auch an den Seitenzahlen unserer Bücher), und so blieb uns nichts anderes übrig, als einige Kapitel bzw. Themen schweren Herzens komplett in das im Preis inbegriffene E-Book auszulagern.

Aus Gründen der Aktualität haben es die Themen

- "Weitere Techniken"
- "Windows Forms-Programmierung"
- "ASP.NET-Programmierung"
- und "Silverlight"

nicht in das gedruckte Buch geschafft. Sie müssen dennoch nicht darauf verzichten, sondern können von allen Vorteilen eines E-Books profitieren (Suchfunktion, Hyperlinks, farbige Abbildungen und nicht zuletzt auch das Gewicht).

Für Sie als Leser bedeutet das: fast 2200 Seiten Material zum Thema "C#-Programmierung", und das zum Preis von **einem** Buch!

Wir hoffen, dass Sie mit der getroffenen Wahl zufrieden sind und beide "Bücher" Ihren Platz auf Ihrem Schreibtisch/PC finden werden.

Walter Doberenz und Thomas Gewinnus *Wintersdorf/Frankfurt/O., im Oktober 2012*

Teil V

Teil V: Weitere Technologien

- **XML in Therorie und Praxis**
- **Einführung in ADO**
- **Das DataSet**
- **OOP-Spezial**
- **Das Microsoft Event Pattern**
- **Verteilen von Anwendungen**
- **Weitere Techniken**
- **Konsolenanwendungen**

Kapitel 22

XML in Theorie und Praxis

Mit XML (*eXtensible Markup Language*) hat sich eine mächtige Metasprache für das Speichern und Austauschen von Daten, insbesondere im Internet, etabliert. Es gibt inzwischen kaum noch eine Anwendung, die nicht mit XML-Unterstützung beworben wird.

Das vorliegende Kapitel kann keinen Anspruch auf Vollständigkeit erheben, sondern nur einen ersten Einblick vermitteln. Weitere Informationen zum Thema finden Sie in den Praxisbeispielen am Ende des vorliegenden Kapitels.

22.1 XML – etwas Theorie

XML ist vor allem auch unter .NET allgegenwärtig, vieles bleibt dabei aber für den Programmierer unsichtbar, da es in Objekten gekapselt ist bzw. von den Tools der Entwicklungsumgebung Visual Studio für Sie automatisch im Hintergrund erledigt wird. Diese Aussage trifft natürlich auch auf die Spracherweiterung *LINQ to SQL* zu, die die Lücke zwischen XML-Daten und Programmiersprache schließt.

22.1.1 Übersicht

Eine der wesentlichsten Ideen, die hinter dem Konzept von XML stehen, ist der Ansatz, Informationen nicht nur darzustellen (wie zum Beispiel in HTML mit seiner festgelegten Syntax), sondern auch deren Inhalt bzw. deren Struktur zu beschreiben. Gleichzeitig soll ein möglichst flexibler und einfacher Datenaustausch zwischen verschiedenen Anwendungen und System-Plattformen möglich sein.

Wie auch bei HTML werden in XML Informationen im Textformat gespeichert. Die einzelnen Elemente des Dokumentes werden durch so genannte Tags gekennzeichnet, diese können ineinander verschachtelt sein.

Im Unterschied zu HTML handelt es sich bei XML jedoch um eine Metasprache, mit deren Hilfe sich neue Sprachen zum Beschreiben von Dokumenten definieren lassen. Die Metasprache XML liefert quasi die grammatikalischen Regeln für den Aufbau von Dokumenten.

HTML, eine klassische Beschreibungssprache, umfasst im Gegensatz dazu lediglich einen festen Satz von vordefinierten Befehlen, der nicht einfach erweiterbar ist. HTML dient im Wesentlichen nur der Darstellung von Informationen.

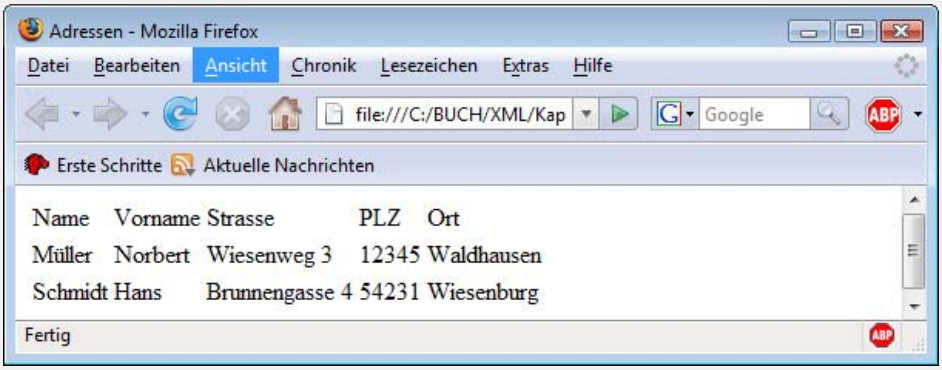
Das einfache Beispiel einer Adressliste soll den Unterschied verdeutlichen. In einer HTML-Datei würden auch Sie die Informationen sicherlich in Form einer Tabelle darstellen.

BEISPIEL 22.1: Adressliste als HTML-Datei

HTML

```
<html><head><title>Adressen</title></head><body>
<table>
  <tr>
    <td>Name</td>
    <td>Vorname</td>
    <td>Strasse</td>
    <td>PLZ</td>
    <td>Ort</td>
  </tr>

  <tr>
    <td>Müller</td>
    <td>Norbert</td>
    <td>Wiesenweg 3</td>
    <td>12345</td>
    <td>Waldhausen</td>
  </tr>
  <tr>
  ...
  </tr>
</table></body></html>
```

Ergebnis



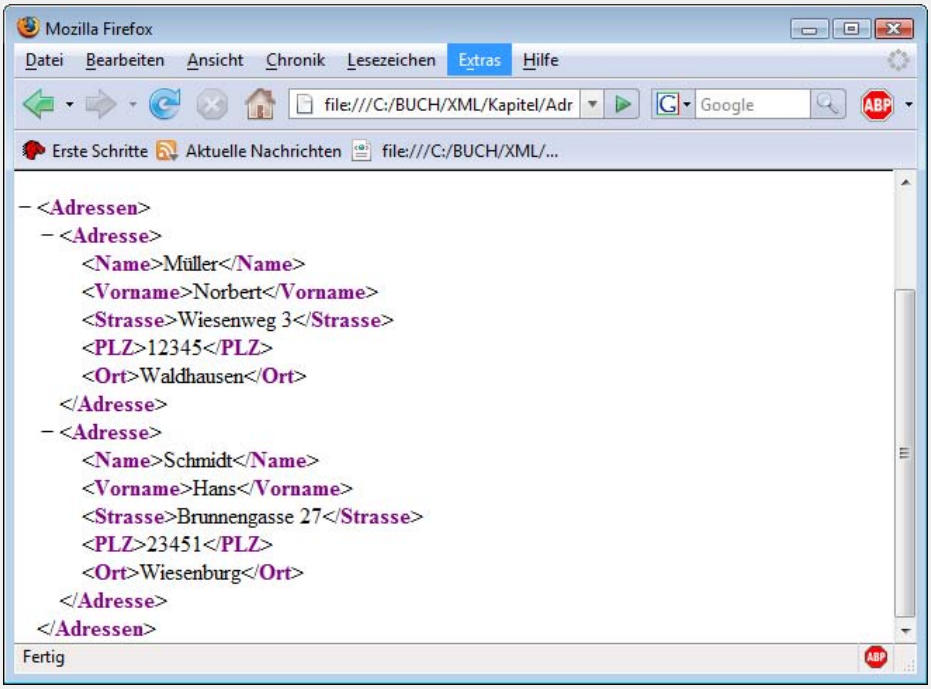
Die obigen Anweisungen sagen lediglich etwas über die Gestaltung der HTML-Seite aus. Einen Zusammenhang zwischen der Tabellenzelle "Name" und dem Inhalt "Müller" kann ein Programm nur schwer herstellen. Eine automatisierte Analyse und Weiterverarbeitung der Datei wird damit fast unmöglich.

BEISPIEL 22.2: Ein einfaches XML-Dokument (Adressverwaltung).

XML

```
<Adressen>
   <Adresse>
      <Name>Müller</Name>
      <Vorname>Norbert</Vorname>
      <Strasse>Wiesenweg 3</Strasse>
      <PLZ>12345</PLZ>
      <Ort>Waldhausen</Ort>
   </Adresse>
   <Adresse>
      <Name>Schmidt</Name>
      <Vorname>Hans</Vorname>
      <Strasse>Brunnengasse 27</Strasse>
      <PLZ>23451</PLZ>
      <Ort>Wiesenburg</Ort>
   </Adresse>
</Adressen>
```

Ergebnis



Wie Sie sehen, steht bei XML die Datenstruktur im Vordergrund. Eine Datenbank-Anwendung, wie zum Beispiel Access, kann relativ leicht feststellen, welche Elemente sich als relationale Tabelle(n) abbilden und importieren lassen.

Im Unterschied zu einer relationalen Datenbank stellt es für ein XML-Dokument kein Problem dar, zum Beispiel einen zweiten Vornamen oder eine zweite Wohnadresse zu speichern.

BEISPIEL 22.3: Ein zusätzlicher Vorname in den Adressdaten

```xml
<Adressen>
   <Adresse>
      <Name>Müller</Name>
      <Vorname>Norbert</Vorname>
      <Vorname>Paul</Vorname>
      <Strasse>Wiesenweg 3</Strasse>
      <PLZ>12345</PLZ>
      <Ort>Waldhausen</Ort>
   </Adresse>
...
   </Adresse>
```

Wie in den obigen Beispielen erkennbar ist, besteht zunächst keine Einschränkung in Bezug auf Anzahl, Länge und Datentyp einzelner Parameter. Es handelt sich lediglich um Textdaten, die von den Tags begrenzt und damit gekennzeichnet werden.

22.1.2 Der XML-Grundaufbau

Wie schon erwähnt, handelt es sich bei XML-Dokumenten um reine Textdateien, die durch Tags strukturiert werden. Innerhalb der XML-Daten lassen sich drei Abschnitte unterscheiden, von denen die beiden ersten optional sind:

- der Prolog mit Steueranweisungen (*Processing Instructions*),
- die DTD (*Document Type Definition*),
- der eigentliche Datenteil.

BEISPIEL 22.4: Prolog

```xml
<?xml version="1.0" standalone="yes" ?>
```

BEISPIEL 22.5: DTD (Definieren von Datentypen)

```xml
<!DOCTYPE DATEN [
<!ELEMENT NAME  (#PCDATA)>
<!ELEMENT VORNAME  (#PCDATA)>
<!ELEMENT ORT  (NAME)>
<!ELEMENT PERSON  (NAME, VORNAME)>
<!ATTLIST ORT marke ID #REQUIRED>
<!ATTLIST PERSON ort IDREF #REQUIRED> ]>
```

BEISPIEL 22.6: Strukturierter Datenteil

```xml
<Adressen>
   <Adresse>
      <Name>Müller</Name>
      <Vorname>Norbert</Vorname>
      <Strasse>Wiesenweg 3</Strasse>
      <PLZ>12345</PLZ>
      <Ort>Waldhausen</Ort>
   </Adresse>
</Adressen>
```

Innerhalb der drei Abschnitte werden

- Elemente,
- Attribute,
- Kommentare

unterschieden.

BEISPIEL 22.7: Ein Element (Vorname ist der Elementname, Norbert ist der Elementinhalt)

```xml
<Vorname>Norbert</Vorname>
```

BEISPIEL 22.8: Ein Attribut (fett hervorgehoben, Hauptwohnsitz ist der Attributname, danach folgt der Wert in Anführungszeichen)

```xml
   <Adresse Hauptwohnsitz="JA">
      <Name>Müller</Name>
      <Vorname>Norbert</Vorname>
   </Adresse>
```

BEISPIEL 22.9: Ein Kommentar (fett hervorgehoben)

```xml
<Adressen>
<!-- ab hier folgen die eigentlichen Daten -->
   <Adresse>
      <Name>Müller</Name>
...
```

Im Folgenden wird im Detail auf die einzelnen Abschnitte bzw. Elemente eingegangen. Doch bevor es so weit ist, wollen wir uns mit den Grundregeln der Sprache XML auseinander setzen.

22.1.3 Wohlgeformte Dokumente

Gerade bei einer so flexiblen Sprache wie XML kommt es darauf an, dass zumindest einige Grundregeln von allen Anwendern eingehalten werden. Nur so ist ein effizientes und schnelles Analysieren der Dokumente möglich. Hält ein Dokument all diese Regeln ein, wird es als *wohlgeformtes*

Dokument bezeichnet. Prüft ein XML-Parser zusätzlich die DTD-Beschreibung oder das XSD-Schema, um die Korrektheit des XML-Dokumentes zu testen, und ist diese Prüfung erfolgreich, bezeichnet man das Dokument als *gültig*. Eine Gültigkeitsprüfung ist jedoch nicht zwingend vorgeschrieben.

Doch zurück zu den Grundregeln für ein wohlgeformtes XML-Dokument. Folgende Regeln müssen Sie in jedem Fall einhalten:

- Jedes Dokument verfügt nur über **ein** Stammelement.
- Start- und Ende-Tags passen zusammen (Groß-/Kleinschreibung beachten!), d h., für jeden Start-Tag ist ein entsprechender Ende-Tag vorhanden. Leere Elemente können mit einem einzelnen Tag dargestellt werden, das mit einem Schrägstrich "/" endet.
- Elemente müssen korrekt geschachtelt sein und dürfen sich nicht überschneiden.
- Die Sonderzeichen <, >, &, ", und ' müssen im Datenteil eines Elements durch *&lt*, *&gt*, *&amp*, *&quot*, *&apos* ersetzt werden.
- Jedes Attribut darf nur einen Wert haben.
- Attributwerte müssen in doppelte oder einfache Anführungszeichen gesetzt werden.
- Die Zeichenfolgen <[[und]]> sind nicht zulässig.

BEISPIEL 22.10: Ein wohlgeformtes XML-Dokument

XML

```
<DATEN>
   <ERDE>
      <KONTINENT>
         <NAME>Europa</NAME>
         <FLÄCHE>10500000</FLÄCHE>
         <EINWOHNER>718500000</EINWOHNER>
         <LAND>
            <NAME>Frankreich</NAME>
            <FLÄCHE>543965</FLÄCHE>
            <EINWOHNER>57800000</EINWOHNER>
         </LAND>

         <LAND>
            <NAME>Deutschland</NAME>
            <FLÄCHE>356854</FLÄCHE>
            <EINWOHNER>80767600</EINWOHNER>
            <ORT marke="01">
               <NAME>Altenburg</NAME>
            </ORT>
            <ORT marke="02" >
               <NAME>Frankfurt Oder</NAME>
            </ORT>
         </LAND>
      </KONTINENT>
```

BEISPIEL 22.10: Ein wohlgeformtes XML-Dokument

```
   </ERDE>
</DATEN>
```

BEISPIEL 22.11: Ein nicht wohlgeformtes Dokument

XML

```
<DATEN>
   <ERDE>
      <KONTINENT>
         <NAME>Europa</NAME>
         <FLÄCHE>10500000</FLÄCHE>
         <EINWOHNER>718500000</EINWOHNER>
         <LAND>
            <NAME>Frankreich</NAME>
            <FLÄCHE>543965</FLÄCHE>
            <EINWOHNER>57800000</EINWOHNER>
         </LAND>
         <LAND>
            <NAME>Deutschland</LAND></NAME>
            <FLÄCHE>356854</FLÄCHE>
            <EINWOHNER>80767600</EINWOHNER>
            <ORT marke=01>
               <NAME>Altenburg</NAME>
            </ORT>
            <ORT marke=02>
               <NAME>Frankfurt Oder</NAME>
            </ORT>
      </KONTINENT>
   </ERDE>
</daten>
<Personen>
      <PERSON>
         <NAME>Müller</NAME>
      </PERSON>
      <PERSON>
         <NAME>Lehmann</NAME>
         <VORNAME>Heinz</VORNAME>
      </PERSON>
</Personen>
```

Die Fehler:

- Start- und Ende-Tag von <DATEN> stimmen nicht überein,
- es sind zwei Stammelemente (*DATEN, Personen*) vorhanden,
- die Attribute (*marke*) sind nicht in Anführungszeichen gesetzt,
- die Tags <LAND> und <NAME> bei Deutschland überschneiden sich.

22.1.4 Processing Instructions (PI)

Für das Verarbeiten von XML-Dokumenten sind so genannte *Processing Instructions* (kurz PI) vorgesehen. PIs werden mit einem Fragezeichen eingeleitet und geschlossen ("<? ... ?>").

Beginnt die PI mit dem Schlüsselwort XML, handelt es sich um eine reservierte XML-Standarddefinition, die dem verarbeitenden Programm (Parser) Informationen über zusätzliche Steuerdateien (DTD), den Zeichensatz und Versionsinformationen liefert.

BEISPIEL 22.12: Einfache Angabe der Versionsnummer

XML
```
<?xml version="1.0"?>
```

BEISPIEL 22.13: Definition eines spezifischen Zeichensatzes über das Schlüsselwort encoding

XML
```
<?xml version="1.0" encoding="UTF-16"?>
```

Jeder XML-Parser muss zumindest die Formate *UTF-8* und *UTF-16* (internationaler Zeichensatz) verarbeiten können.

Möchten Sie dem Parser mitteilen, dass eine externe DTD vorhanden ist, können Sie dies mit der Option *standalone* realisieren. Gültige Werte sind

- *yes* (keine externe DTD vorhanden) und
- *no* (eine externe DTD ist vorhanden, der Dateiname steht in der *DOCTYPE*-Definition).

BEISPIEL 22.14: Keine externe DTD vorhanden

XML
```
<?xml version="1.0" standalone="yes" ?>
```

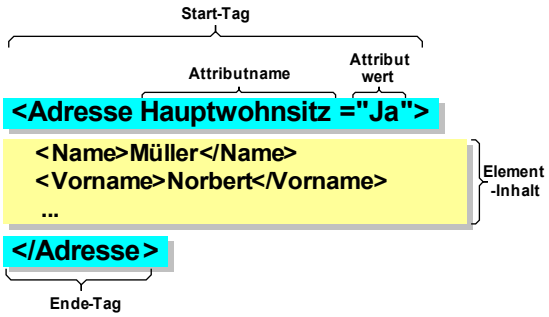
BEISPIEL 22.15: Eine externe DTD (welt.dtd) ist vorhanden

XML
```
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE WELT SYSTEM "welt.dtd">
```

HINWEIS: Mittlerweile haben sich die XSD-Schemas gegenüber der DTD weitgehend durchgesetzt, wir gehen deshalb im Rahmen dieses Kapitels nicht weiter auf die DTD ein.

22.1.5 Elemente und Attribute

Wie schon erwähnt, bestehen Elemente aus einem Start- und einem Ende-Tag. Beide Tags schließen den Inhalt (dies kann Text sein, aber auch ein bzw. mehrere Elemente) ein. Die folgende Abbildung zeigt den Elementeaufbau:



Jedes Element muss den folgenden Regeln genügen:

- Der Elementname beginnt mit einem Buchstaben oder einem Unterstrich.
- Nach dem ersten Zeichen können beliebige Zeichen folgen.
- Elementnamen berücksichtigen die Groß-/Kleinschreibung.
- Elementnamen dürfen keine Leerzeichen enthalten.
- Start- und Ende-Tag müssen in der Schreibweise übereinstimmen.
- Leere Elemente, auch als "Singleton" bezeichnet, können mit einem Tag (z.B. <Adresse/> statt <Adresse></Adresse>) gekennzeichnet werden.

Attribute sind zusätzliche Eigenschaften, die dem Start-Tag hinzugefügt werden können, um den Inhalt näher zu spezifizieren. Wie auch bei den Elementnamen wird zwischen Groß-/Kleinschreibung unterschieden, Attributwerte müssen in doppelten Anführungszeichen eingeschlossen werden. Ein Attribut darf nur einen Wert haben.

BEISPIEL 22.16: Falsche Deklaration

```xml
<Anschrift Name="Müller" Vorname="Norbert" Vorname="Hans">
```

In diesem und in den meisten anderen Fällen ist es günstiger, Attribute als untergeordnete Elemente darzustellen.

BEISPIEL 22.17: Richtige Deklaration

```xml
<Anschrift>
  <Name>Müller</Name>
  <Vorname>Norbert</Vorname>
  <Vorname>Hans</Vorname>
  <Vorname>Werner</Vorname>
</Anschrift>
```

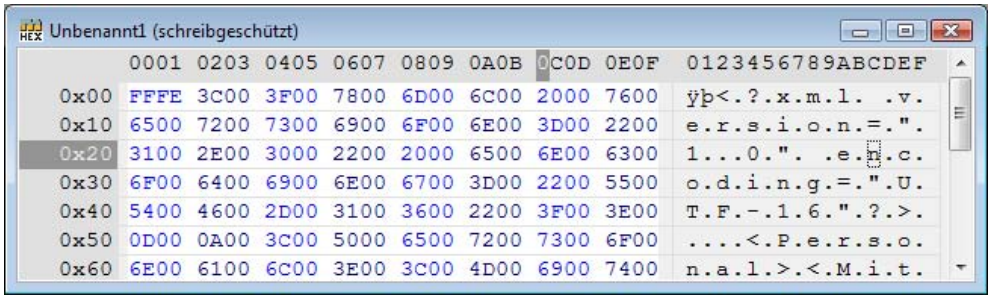
HINWEIS: Beachten Sie die Vorschrift, dass jedes XML-Dokument über genau ein Wurzel-Element (Root) verfügen muss.

22.1.6 Verwendbare Zeichensätze

In den vorhergehenden Abschnitten war ja bereits kurz die Rede von Zeichensätzen und in diesem Zusammenhang fiel auch die Bezeichnung "UTF". Hierbei handelt es sich um die Abkürzung für *Unicode Transformation Format*.

Diese Formate bilden die Zeichenfolgen der XML-Daten als eindeutige Byte-Sequenzen ab. Zur eindeutigen Identifikation wird an den Beginn der Datei/des Datenstroms eine spezielle Identifikationsmarke, die *BOM (Byte Oder Mark)* eingefügt. Ein Blick in eine UTF-16-XML-Datei zeigt, was gemeint ist:

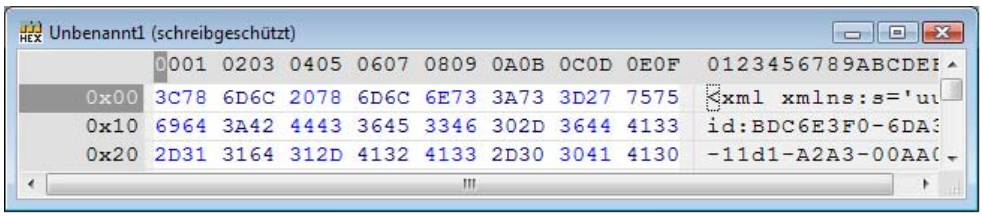
Eine UTF-16-Datei im Hex-Editor:



Während Sie zum Beispiel in Notepad oder im Internet Explorer nur die gewohnten Processing Instructions am Beginn des Dokuments vorfinden, zeigt der Hex-Editor zwei zusätzliche Bytes am Dateianfang "FFFE". Dabei handelt es sich um die BOM für eine UTF-16-Kodierung im LE-Format[1] (*Little-Endian*).

HINWEIS: Alternativ könnte hier auch "FEFF" stehen, was auf eine UTF-16-Kodierung im BE-Format (*Big-Endian*) hinweist.

Sehen Sie sich dagegen z.B. eine mit den ADO (*Recordset.Save*) erzeugte XML-Datei an, werden Sie keine BOM vorfinden:



Diese im UTF-8-Format gespeicherte Datei kann auf BOM und explizite Formatangaben in den Processing Instructions verzichten, da es sich automatisch um das Standard-XML-Format UTF-8

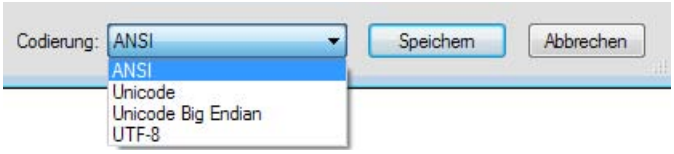
[1] Bei Unicode-Zeichen kann das höherwerte Byte das erste Byte (Little-Endian) oder das zweite Byte (Big-Endian) sein.

handelt. Gleiches trifft auf eine im ASCII-Format gespeicherte XML-Datei zu (die Zeichen bis 127 sind in ASCII und UTF-8 identisch).

HINWEIS: Im Gegensatz zu UTF-8 besitzt eine UTF-16 kodierte Datei immer einen BOM!

Doch was ist, wenn eine reine ASCII-XML-Datei beispielsweise einen Umlaut enthält? Spätestens hier zeigt Ihnen der Internet Explorer beim Laden die "Rote Karte":

Der Umlaut (in diesem Fall "ü") wird mit "FC" kodiert, dies entspricht jedoch keinem zulässigen UTF-8-Code. UTF-8 verwendet in diesem Fall zwei Bytes zur Kodierung des betreffenden Zeichens (C3BC). Beachten Sie dies, wenn Sie eine XML-Datei z.B. per Notepad erzeugen. Hier müssen Sie im Speichern-Dialog die richtige Kodierung auswählen:



Auch beim Programmieren mit DOM oder ADO.NET können Sie schnell über die Fallstricke der Kodierung stolpern, wenn Sie zum Beispiel mit Streams arbeiten und zusätzlich direkten Einfluss auf die Processing Instructions nehmen. Tritt hier eine Diskrepanz zwischen der in den Processing Instructions genannten Kodierung und der tatsächlich verwendeten Kodierung (bzw. der BOM) auf, ist die Datei später nicht mehr lesbar.

Doch wann sollte welches Format verwendet werden?

Eine allgemein gültige Antwort kann auf die Schnelle nicht gegeben werden, da sich aus der Verwendung der verschiedenen Kodierungen auch Vor- bzw. Nachteile ergeben:

Nutzen Sie die UTF-8-Kodierung, wenn es sich um normale Tabellendaten handelt. Da hier im Normalfall nur ein Zeichen (bzw. zwei bei Umlauten) verwendet werden, sparen Sie gegenüber einer UTF-16-Kodierung die Hälfte an Speicherplatz ein. Doch Vorsicht: Enthält eine UTF-8-Datei

viele Zeichen mit einem Code über einem Byte, wird die Datei schnell drei mal so groß (zum Beispiel bei asiatischen Zeichen). Hier ist UTF-16 die bessere Wahl.

22.2 XSD-Schemas

Bei der Arbeit mit .NET ist Ihnen sicher schon mehrfach der Begriff *XSD-Schemas* untergekommen. Worum handelt es sich eigentlich? Zunächst einmal wird mit XSD die *XML Schema Definition Language* bezeichnet.

Ein XSD-Schema ist ein eigenständiges Dokument (oder auch ein zusätzlicher Teil des XML-Dokuments), das die Struktur der XML-Daten beschreibt. Grundsätzlich können Sie also ein XSD-Schema mit den Strukturinformationen z.B. einer Access- oder SQL-Server-Datenbank vergleichen.

Ein Schema kann folgende Informationen und Vorgaben enthalten:

- Die Definition der einzelnen Datentypen für die XML-Elemente und -Attribute
- Diverse Einschränkungen (z.B. Eindeutigkeit (Unique) von Werten)
- Zusammenhänge zwischen einzelnen XML-Elementen (Relationen)

Nur beim Vorliegen von XML-Daten **und** Schema-Informationen können also logische Zusammenhänge hergestellt werden, ohne die ein sinnvolles Auslesen der XML-Daten nicht möglich ist.

22.2.1 XSD-Schemas und ADO.NET

ADO.NET nutzt die Schemas, um einen Zusammenhang zwischen der rein hierarchischen Struktur der XML-Dokumente und der relationalen Struktur innerhalb der DataSets herzustellen. Erzeugen Sie ein neues DataSet, sind zunächst keinerlei Tabellen definiert. Diese Informationen können Sie entweder aus einer XSD-Datei laden oder Sie erzeugen diese Informationen über Methodenaufrufe.

BEISPIEL 22.18: Ein einfaches Schema für zwei Tabellen, in denen Telefondaten für Mitarbeiter gespeichert werden können

XML

```
<?xml version="1.0" standalone="yes" ?>
<xs:schema id="Telefon" targetNamespace="http://www.tempuri.org/Telefon.xsd"
xmlns:mstns="http://www.tempuri.org/Telefon.xsd" xmlns="http://www.tempuri.org/Telefon.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:schemas-microsoft-com:xml-
msdata" attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="Telefon" msdata:IsDataSet="true" msdata:Locale="de-DE">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
```

Hier wird die erste Tabelle definiert:

```
        <xs:element name="Mitarbeiter">
          <xs:complexType>
```

BEISPIEL 22.18: Ein einfaches Schema für zwei Tabellen, in denen Telefondaten für Mitarbeiter gespeichert werden können

Die Definition der Tabellenspalten (Attribute):

```
        <xs:attribute name="ID" form="unqualified" msdata:AutoIncrement="true"
        type="xs:int" />
          <xs:attribute name="Vorname" form="unqualified" type="xs:string" />
          <xs:attribute name="Nachname" form="unqualified" type="xs:string" />
        </xs:complexType>
      </xs:element>
```

Die zweite Tabelle:

```
      <xs:element name="Telefone">
        <xs:complexType>
          <xs:attribute name="MitarbeiterId" form="unqualified" type="xs:int" />
          <xs:attribute name="Nummer" form="unqualified" type="xs:string" />
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
```

Definition eines Primärschlüssels:

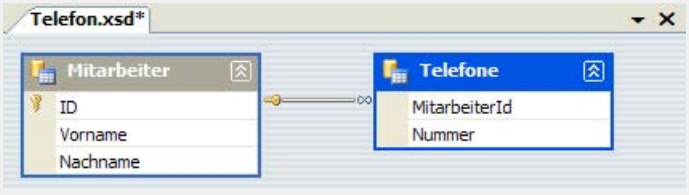
```
  <xs:key name="TelefonKey1" msdata:PrimaryKey="true">
    <xs:selector xpath=".//mstns:Mitarbeiter" />
    <xs:field xpath="@ID" />
  </xs:key>
```

Festlegen des Zusammenhangs zwischen beiden Tabellen (Relation):

```
  <xs:keyref name="MitarbeiterTelefone" refer="mstns:TelefonKey1">
    <xs:selector xpath=".//mstns:Telefone" />
    <xs:field xpath="@MitarbeiterId" />
  </xs:keyref>
 </xs:element>
</xs:schema>
```

Ergebnis

Die folgende Abbildung zeigt das relationale Abbild des obigen Schemas:



BEISPIEL 22.19: Ein auf dem vorhergehenden Schema aufbauendes XML-Dokument

XML
```
<?xml version="1.0" encoding="utf-8" ?>
<Telefon xmlns="http://www.tempuri.org/Telefon.xsd">
  <Mitarbeiter ID="0" Vorname="Thomas" Nachname="Gewinnus"></Mitarbeiter>
  <Mitarbeiter ID="1" Vorname="Walter" Nachname="Doberenz"></Mitarbeiter>
  <Telefone MitarbeiterId="0" Nummer="0335-1234567"></Telefone>
  <Telefone MitarbeiterId="0" Nummer="0172-888777666"></Telefone>
  <Telefone MitarbeiterId="1" Nummer="0345-12345678"></Telefone>
  <Telefone MitarbeiterId="1" Nummer="0171-111222333"></Telefone>
</Telefon>
```

Wie Sie sehen, werden zum Speichern der Informationen (*Nachname*, *Vorname*) Attribute statt Elemente verwendet. Auf diese Weise ist das Dokument etwas kompakter.

Den Elementen *Mitarbeiter* sind je zwei (beliebig viele) Elemente *Telefone* zugeordnet, die klassische 1:n-Beziehung aus der relationalen Welt.

22.2.2 XML-Schemas in Visual Studio analysieren

Vielleicht kann sich der eine oder andere noch daran erinnern, dass es in Visual Studio 2005 mal einen XSD-Designer gegeben hat, mit dem man auf recht einfache Weise ein XSD-Schema erstellen konnte. Bevor Sie sich jetzt in der Visual Studio-IDE auf die Suche machen, gleich ein wichtiger Hinweis vorweg[1]:

HINWEIS: Visual Studio 2012 verfügt, wie auch die Vorgängerversion 2010, über **keinen** grafischen Editor für XSD-Schemas, auch wenn die Bezeichnung einiger Menüpunkte mit "XML-Schema-Designer" etwas anderes suggeriert!

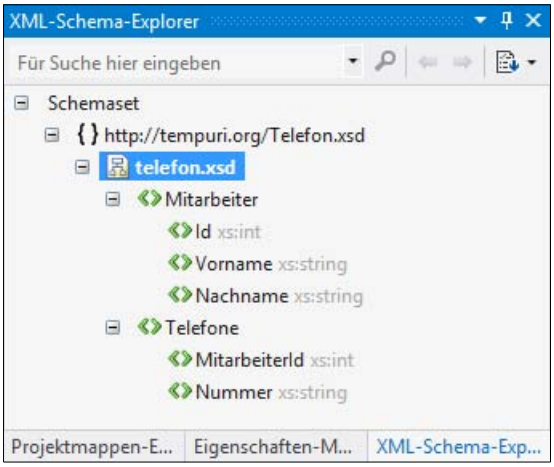
Doch was bietet Visual Studio 2012 dem Entwickler in Bezug auf XML-Schemas? Im Wesentlichen handelt es sich um verschiedene Varianten von Betrachtern, mit denen Sie vorhandene Schemadefinitionen grafisch ansprechend darstellen können:

- einen XML-Schema-Explorer
- den XML-Schema-Designer (der keiner ist)
- die Inhaltsmodellansicht (*Content Model View*)
- die Diagrammansicht (*Graph View*)

[1] Da wir uns in der Vorgängerversion auf eine Aussage von Microsoft verlassen hatten ("der Schema Designer wird nachgereicht ...") möchten wir an dieser Stelle gleich für Klarheit sorgen.

Der XML-Schema-Explorer

Öffnen Sie eine XML-Schemadatei im XML-Editor oder im XML-Schema-Designer, so wird der XML-Schema-Explorer geöffnet.



HINWEIS: Ist der Schema Explorer nicht zu sehen, können Sie ihn über den Menüpunkt *Ansicht|XML Schema Explorer* einblenden.

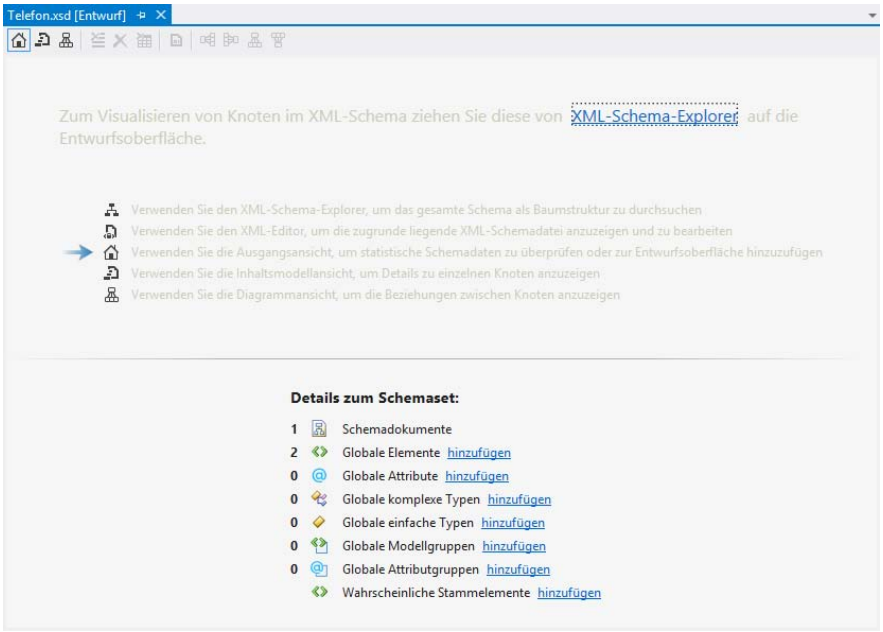
In einer hierarchischen Strukturansicht werden neben dem aktuellen Schema auch alle zugehörigen XSD-Dateien, d h. eingeschlossene oder importierte Schemas, angezeigt. Zu besseren Übersicht lassen sich Details ein-/ausblenden, Sortierfolgen ändern oder direkt Einträge suchen.

HINWEIS: Klicken Sie doppelt auf einen der Einträge im XML-Schema-Explorer, so wird bei geöffnetem XML-Editor der entsprechende Eintrag markiert. Möchten Sie aus dem XML-Editor heraus einen Eintrag im XML-Schema-Explorer anzeigen, verwenden Sie dazu den entsprechenden Kontextmenüeintrag des Editors

Ausgangsansicht des XML-Schema-Designers

Öffnen Sie eine XML-Schema-Datei, so öffnet sich standardmäßig die Ausgangsansicht des XML-Schema-Designers, in der Sie zunächst die anzuzeigenden Details auswählen müssen, bevor Sie sich an den grafischen Darstellungen erfreuen können.

Die Auswahl der Elemente/Attribute etc. erfolgt über die Hyperlinks am unteren Ende der Seite, oder Sie verwenden den XML-Schema-Explorer, um die gewünschten Details per Drag & Drop in den Designer einzufügen.

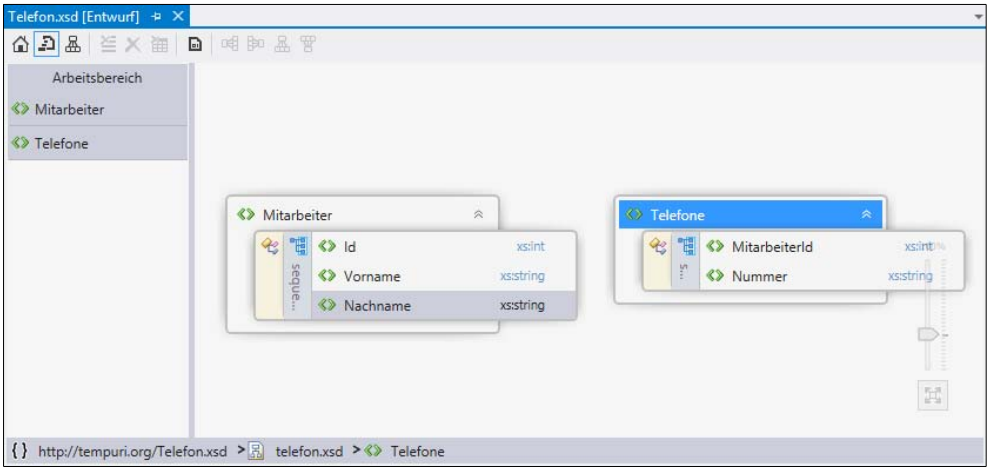


HINWEIS: An dieser Stelle sieht man auch wieder mal wie sinnvoll das "Trauerlayout" von Visual Studio 2012 ist, die Schrift (Grau auf Grau) ist kaum zu erkennen.

Nachfolgend können Sie in die Inhaltsmodellansicht oder in die Diagrammansicht wechseln.

Inhaltsmodellansicht

In dieser Ansicht können Sie die in Ihrem Schema enthaltenen Informationen in übersichtlicher Form darstellen und Detaildaten ein-/ausblenden:



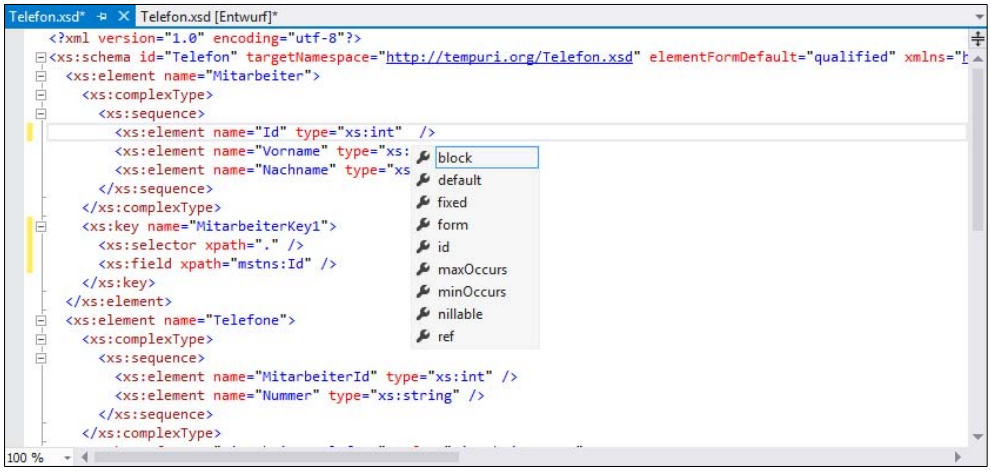
HINWEIS: Weitere Detaildaten, wie zum Beispiel Min/Max-Werte oder Default-Werte können Sie im Eigenschaftenfenster einsehen.

Die erzeugten Ansichten lassen sich als Bild exportieren, weiterhin lassen sich über das Kontext-Menü auch Beispiel-XML-Daten wie diese generieren:

```
<?xml version="1.0" encoding="utf-8"?>
<Telefon xmlns="http://www.tempuri.org/Telefon.xsd">
  <Mitarbeiter ID="1" Vorname="Vorname1" Nachname="Nachname1" />
  <Telefone MitarbeiterId="1" Nummer="Nummer1" />
  <Mitarbeiter ID="-2147483647" Vorname="Vorname2" Nachname="Nachname2" />
</Telefon>
```

XML-Editor

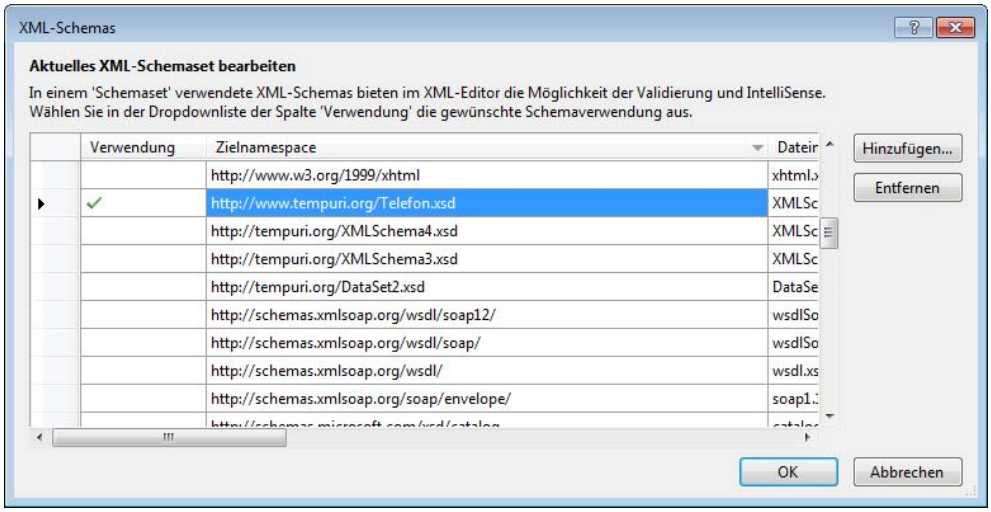
Last but not least, gibt es natürlich auch eine Möglichkeit, Schema-Dateien zu editieren, doch hier haben wir es mit einem recht einfachen XML-Texteditor zu tun, umfangreiche Schemadefinitionen werden Sie mit diesem Editor wohl kaum erstellen wollen, die Übersicht leidet sicher schon nach wenigen Zeilen, und der XML-Schema-Explorer gibt auch keine große Hilfestellung, ganz zu schweigen davon, dass Sie auch detaillierte XSD-Kenntnisse benötigen.



22.2.3 XML-Datei mit XSD-Schema erzeugen

Erstellen Sie eine neue XML-Datei (Menüpunkt *Projekt/Neues Element hinzufügen...*). Nach dem Öffnen dieser Datei können Sie die Eigenschaften des Dokuments bearbeiten. Für uns ist die Eigenschaft *Schemata* interessant. Weisen Sie hier das bereits erstellte XSD-Schema zu[1]:

[1] Dazu müssen Sie beim gewünschten Schema in der ersten Spalte der Liste ein Häkchen setzen.



HINWEIS: Alternativ können Sie auch eines des vordefinierten Schemata im eingeblendeten Assistenten auswählen.

Im Anschluss können Sie das Dokument im Editor bearbeiten:

22.2.4 XSD-Schema aus einer XML-Datei erzeugen

Nicht in jedem Fall liegt einer XML-Datei auch gleich ein entsprechende Schema bei, doch in vielen Anwendungsfällen wird dieses benötigt. Bevor Sie jetzt anfangen, mühsam die Struktur der XML-Daten zu entziffern, werfen Sie ruhig einmal einen Blick auf den Menüpunkt *XML*[1], der eine entsprechende Funktionalität bereitstellt. Alternativ können Sie auch das Tool *xsd.exe* aus dem .NET-SDK verwenden oder Sie nutzen die Möglichkeit, das Schema zur Laufzeit mittels Methodenaufrufen zu generieren. Wir kommen im weiteren Verlauf des Kapitels darauf zurück.

[1] Dieser erscheint, wenn Sie eine XML-Datei öffnen.

22.3 Verwendung des DOM unter .NET

Obwohl es unter .NET auch einfachere Wege zum Erzeugen/Verarbeiten von XML-Dateien gibt, wollen wir Sie zunächst mit der "Low-Level"-Programmierung vertraut machen. Zum einen bietet sich hier die Möglichkeit, das in den vorhergehenden Abschnitten Gelernte "hautnah" nachzuvollziehen, zum anderen bekommen Sie so ein besseres Verständnis für die diversen Optionen bei der XML-Ausgabe.

22.3.1 Übersicht

DOM steht für *Document Object Model* und definiert ein Objektmodell, mit dem sich XML-Dokumente plattformübergreifend bearbeiten lassen. Plattformübergreifend können Sie in diesem Fall auch wirklich gelten lassen, denn DOM ist ein allgemein gültiger Standard des W3C-Konsortiums und keine Exklusiv-Erfindung von Microsoft.

DOM erspart Ihnen den steinigen Weg, eine XML-Datei als Textdatei zu verarbeiten, d h. mühsam zu parsen und zu filtern. Alle XML-Elemente werden durch das DOM als Eigenschaften bzw. Collections von Objekten abgebildet. Elemente lassen sich über Methoden erzeugen, modifizieren und löschen. Die XML-Daten werden als Baum mit einzelnen Zweigen dargestellt. Weiterhin können Sie über das DOM auch allgemeine Informationen über das Dokument abfragen.

Die wichtigsten Knotentypen, die mit dem DOM angesprochen werden können:

- *Document Type*
- *Processing Instruction*
- *Element*
- *Attribute*
- *Text*

Die folgende Tabelle gibt Ihnen eine kurze Übersicht über die wichtigsten Objekte:

Objekt	Beschreibung
XMLDocument	Dieses Objekt repräsentiert das gesamte XML-Dokument bzw. dessen Hauptknoten
XMLNode	Dieses Objekt repräsentiert einen einzelnen Knoten innerhalb des XML-Baums
XMLNodeList	Collection von *XMLNode*-Objekten
XMLNamedNodeMap	Ebenfalls eine Collection von *XMLNode*-Objekten, mit denen der Zugriff auf die Attribute möglich ist
XMLParseError	Objekt zur gezielten Fehleranalyse, das neben der Fehlerbeschreibung auch die Position und die Fehlernummer bereitstellt
XMLAttribute	Objekt zum Zugriff auf Element-Attribute

Objekt	Beschreibung
XMLCDATASection	Objekt für den Zugriff auf CDATA-Abschnitte (diese werden nicht vom Parser verarbeitet)
XMLCharacterData	Objekt für Textmanipulationen
XMLComment	Objekt für Zugriff auf Kommentare
XMLDocumentType	Objekt für den Zugriff auf DTD
XMLElement	Dieses Objekt repräsentiert ein Element
XMLEntity	Dieses Objekt repräsentiert eine Entität
XMLImplementation	Über die Methode *HasFeature* können Informationen über die DOM-Implementation angefragt werden
XMLProcessingInstruction	Dieses Objekt repräsentiert eine Processing Instruction (PI)
XMLText	Der textuelle Inhalt eines Elements oder eines Attributes

22.3.2 DOM-Integration in C#

Selbstverständlich ist auch in C# eine Zugriffsmöglichkeit auf das DOM integriert. Über den Namespace *System.XML* importieren Sie alle relevanten Objekte und Klassen für die Arbeit mit XML-Dateien.

Ausgangspunkt aller weiteren Experimente ist das *XMLDocument*-Objekt, mit dem Sie den Zugriff auf ein XML-Dokument realisieren können. Eine Instanz bilden Sie einfach durch einen parameterlosen Konstruktor.

HINWEIS: An dieser Stelle möchten wir lediglich mit einigen Beispielen die Verwendung der DOM-Objekte demonstrieren, für eine vollständige Auflistung aller Eigenschaften und Methoden fehlt einerseits der Platz, andererseits sollte dies besser einem Buch über XML vorbehalten bleiben.

22.3.3 Laden von Dokumenten

Haben Sie erfolgreich eine Instanz erzeugt, können Sie auch schon ein neues XML-Dokument generieren oder eine vorhandene Datei von der Festplatte oder aus dem Internet laden. Die *LoadXml*-Methode unterstützt sowohl "normale" Pfadangaben (*\\server\e\files\test.xml* bzw. *c:\test.xml*) als auch URL-Angaben mit Webadressen (*http://www.xyz-abc.com/test.xml*).

BEISPIEL 22.20: Laden aus einer Datei

C#
```
XmlDocument xmldoc = new XmlDocument();
try
{
  xmldoc.LoadXml("C:\test.xml");
  MessageBox.Show("OK");
}
```

BEISPIEL 22.20: Laden aus einer Datei

```
    catch
    {
      MessageBox.Show("Fehler");
    }
```

Alternativ können Sie die Dokumente auch aus einer *String*-Variablen laden. Verwenden Sie dazu die *LoadXML*-Methode.

BEISPIEL 22.21: Laden von XML-Daten aus einem String

C#

```
    XmlDocument xmldoc = new XmlDocument();
    String mystr;

    mystr = "<WELT>";
    mystr = mystr + "  <KONTINENTE>";
    mystr = mystr + "  </KONTINENTE>";
    mystr = mystr + "</WELT>";
    try
    {
      xmldoc.LoadXml(mystr);
      MessageBox.Show("OK");
    }
    catch
    {
     MessageBox.Show("Fehler");
    }
```

HINWEIS: Im Gegensatz zum DOM-Zugriff über COM-Objekte werden durch die Methoden keine *true/false*-Werte zurückgegeben. Sie müssen also mit *try catch* arbeiten. Das *ParseError*-Objekt ist in diesem Zusammenhang auch "entsorgt" worden.

22.3.4 Erzeugen von XML-Dokumenten

Bevor wir uns mit der Verarbeitung vorhandener XML-Dokumente beschäftigen, wollen wir zunächst selbst "Hand anlegen" und neue XML-Dokumente erzeugen.

BEISPIEL 22.22: Ein erstes XML-Dokument (Versuch 1)

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root;
```

Wir erzeugen ein Wurzel-Element ...

```
    root = xmldoc.CreateElement("WELT");
```

BEISPIEL 22.22: Ein erstes XML-Dokument (Versuch 1)

... und fügen dieses dem DOM-Dokument hinzu:

```
xmldoc.AppendChild(root);
MessageBox.Show(xmldoc.InnerXml);
```

Ergebnis

Klein, aber fein präsentiert sich unser erstes XML-Dokument in der Messagebox:

Basierend auf den Erkenntnissen des vorhergehenden Beispiels erweitern wir das Programm so, dass das Element *WELT* weitere untergeordnete Elemente erhält. Das erzeugte XML-Dokument soll in diesem Fall als Datei gesichert werden.

BEISPIEL 22.23: Unser zweiter Versuch (Sichern in Datei)

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root, node;
```

Wir erzeugen ein Wurzel-Element ...

```
root = xmldoc.CreateElement("WELT");
```

... und fügen dieses dem DOM-Dokument hinzu:

```
xmldoc.AppendChild(root);
```

Ähnlich wie bei einem *TreeView*-Control werden nun dem bereits erzeugten Root-Knoten weitere Untereinträge hinzugefügt:

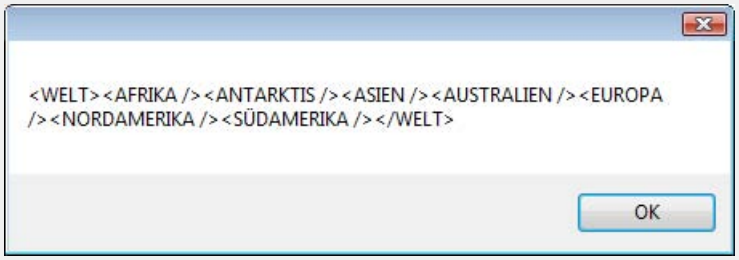
```
node = xmldoc.CreateElement("AFRIKA");
root.AppendChild(node);
node = xmldoc.CreateElement("ANTARKTIS");
root.AppendChild(node);
node = xmldoc.CreateElement("ASIEN");
root.AppendChild(node);
node = xmldoc.CreateElement("AUSTRALIEN");
root.AppendChild(node);
node = xmldoc.CreateElement("EUROPA");
root.AppendChild(node);
node = xmldoc.CreateElement("NORDAMERIKA");
root.AppendChild(node);
node = xmldoc.CreateElement("SÜDAMERIKA");
root.AppendChild(node);
```

BEISPIEL 22.23: Unser zweiter Versuch (Sichern in Datei)

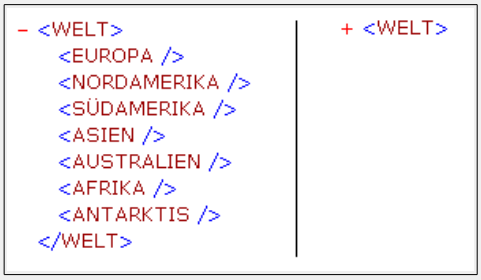
Wir sichern die XML-Daten in einer Datei:

```
xmldoc.Save(@"c:\Test2.xml");
MessageBox.Show(xmldoc.InnerXml);
```

Ergebnis



Die Anzeige im Internet Explorer ist dagegen wesentlich aufschlussreicher (geöffnete und geschlossene Ansicht), stellt doch der Explorer automatisch die hierarchische Struktur der XML-Daten dar.



22.3.5 Auslesen von XML-Dateien

Nachdem wir bereits eine erste einfache XML-Datei erzeugt haben, können wir diese auch laden und anzeigen lassen. Dazu benötigen wir in jedem Fall neben dem bekannten *XMLDocument*-Objekt auch ein *XMLNode*-Objekt, um zumindest auf das Wurzel-Element der XML-Daten zugreifen zu können.

BEISPIEL 22.24: Bestimmen des Wurzel(Root)-Elements, Einlesen der eben erzeugten Datei und Anzeige des ersten Elements.

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root, node;

    xmldoc.Load(@"c:\test2.xml");
```

Einen "Zeiger" auf das Wurzelelement ("WELT") bestimmen:

```
    root = xmldoc.DocumentElement;
```

BEISPIEL 22.24: Bestimmen des Wurzel(Root)-Elements, Einlesen der eben erzeugten Datei und Anzeige des ersten Elements.

Nachfolgend können wir den ersten Untereintrag ermitteln und anzeigen:

```
node = root.FirstChild;
MessageBox.Show(node.Name);
```

Ergebnis

Die Ausgabe lautet "AFRIKA", entsprechend der Reihenfolge, in der die Elemente im vorhergehenden Beispiel gesichert wurden:

Für den Zugriff auf die weiteren Elemente gibt es mehrere Varianten:

- Entweder Sie lesen die Elemente über die *ChildNodes*-Collection des Wurzel-Elementes aus,
- oder Sie "hangeln" sich mit der Methode *NextSibling* durch die Objektliste,
- oder Sie sind ein ganz konventioneller Programmierer, d h., Sie verwenden die gute alte *for*-Schleife.

Das Ergebnis ist in allen Fällen dasselbe.

BEISPIEL 22.25: Anzeige der einzelnen Elemente mit Hilfe der DOM-Eigenschaften und Methoden in einer ListBox.

C#

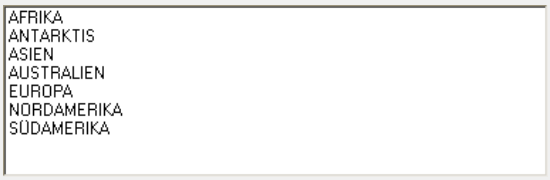
```
XmlDocument xmldoc = new XmlDocument();
XmlNode root;
      xmldoc.Load(@"c:\test2.xml");
      listBox1.Items.Clear();
      root = xmldoc.DocumentElement;
```

Auslesen mit Hilfe der Collection *ChildNodes*:

```
      foreach (XmlNode node in root.ChildNodes)
        listBox1.Items.Add(node.Name);
```

Ergebnis

Die Ausgabe in der *ListBox:*



22.3.6 Direktzugriff auf einzelne Elemente

Sicher dürfte es recht mühsam sein, bei der Suche nach einem bestimmten Baumelement immer gleich die ganze Liste zu durchforsten, um zum Beispiel das Element *EUROPA* zu finden. In diesem Fall hilft Ihnen die *SelectSingleNode*-Methode weiter.

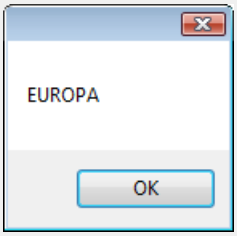
BEISPIEL 22.26: Verwendung von *SelectSingleNode*

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root, node;

xmldoc.Load(@"c:\test2.xml");
root = xmldoc.DocumentElement;
node = root.SelectSingleNode("EUROPA");
MessageBox.Show(node.Name);
```

Ergebnis

Erwartungsgemäß wird Ihnen der zugehörige Knotenname angezeigt:



Über das zurückgegebene Objekt vom Typ *XMLNode* können Sie wiederum auf die jeweiligen Untereinträge des Knotens zugreifen usw. Doch bevor wir dies demonstrieren, möchten wir unsere Datenbasis etwas vergrößern und zusätzliche Informationen im XML-Baum abspeichern.

22.3.7 Einfügen von Informationen

Nachdem Sie die XML-Daten geladen und einen bestimmten Knoten ausgewählt haben, können Sie diesem zusätzliche Informationen in Form von Attributen oder Elementen hinzufügen.

BEISPIEL 22.27: Attribute und Elemente verwenden

C#

Den einzelnen Kontinenten, die bereits in der Datei enthalten sind, ordnen wir noch die Fläche, die Anzahl der Einwohner und einige Länder zu. Dabei werden wir uns jedoch auf einige Auszüge beschränken, für eine komplette Auflistung dürfte an dieser Stelle weder der Platz noch der Bedarf vorhanden sein.

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root, node;
xmldoc.Load(@"c:\test2.xml");
```

BEISPIEL 22.27: Attribute und Elemente verwenden

Auswahl eines bestimmten Knotens:

```
root = xmldoc.DocumentElement;
node = root.SelectSingleNode("EUROPA");
```

Einfügen von Zusatzinformationen:

```
node.AppendChild(xmldoc.CreateElement("Fläche")).InnerText = "10500000";
node.AppendChild(xmldoc.CreateElement("Einwohner")).InnerText = "718500000";
```

Wie Sie sehen, verwenden wir eine recht kurze Schreibweise für den Zugriff auf den zurückgegebenen *ChildNode*. Sie könnten auch umständlich den Rückgabewert einer Variablen von Typ *XMLNode* zuweisen und nachfolgend auf die Eigenschaften zugreifen.

Einfügen weiterer Elemente:

```
node.AppendChild(xmldoc.CreateElement("Frankreich"));
node.AppendChild(xmldoc.CreateElement("Deutschland"));
node.AppendChild(xmldoc.CreateElement("Italien"));
node.AppendChild(xmldoc.CreateElement("Österreich"));
node.AppendChild(xmldoc.CreateElement("Schweden"));
node.AppendChild(xmldoc.CreateElement("Norwegen"));
node.AppendChild(xmldoc.CreateElement("Polen"));
```

Auch hier nutzen wir die Möglichkeit, den zurückgegebenen Wert (es handelt sich um ein Objekt vom Typ *XMLNode*) gleich an die nächste Methode weiterzugeben. So ersparen wir uns unnötige Variablen und natürlich auch einige Zeilen Quellcode.

```
xmldoc.Save(@"c:\Test6.xml");
...
```

Ergebnis

Das Aussehen der Datei nach diesen Erweiterungen zeigt die folgende Abbildung:

Möchten Sie sich nicht umständlich durch die Objektstruktur hangeln, können Sie auch direkt den Pfad zum gesuchten Element angeben.

BEISPIEL 22.28: Erweitern der Einträge von Deutschland und Frankreich um die Einträge *Fläche* und *Einwohner*

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root, node;

xmldoc.Load(@"c:\test6.xml");
root = xmldoc.DocumentElement;
```

Wir bestimmen direkt aus der Root heraus den gesuchten Knoten ...

```
node = root.SelectSingleNode("EUROPA/Deutschland");
```

... und fügen auf bewährte Weise zwei neue Einträge hinzu:

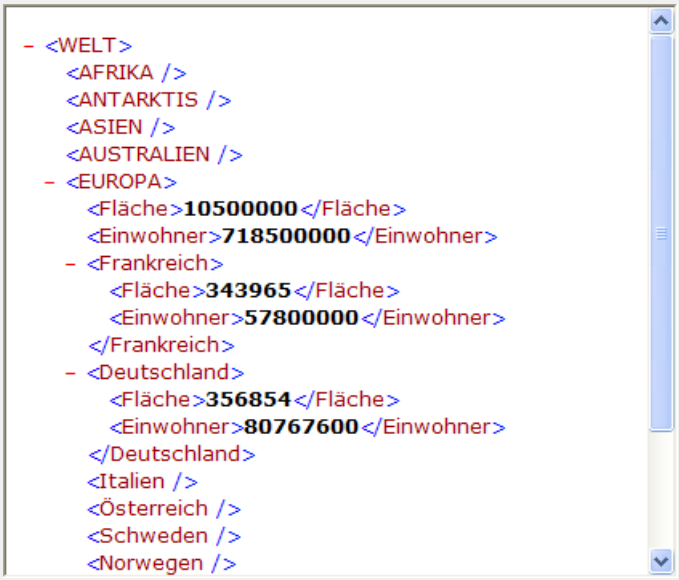
```
node.AppendChild(xmldoc.CreateElement("Fläche")).InnerText = "356854";
node.AppendChild(xmldoc.CreateElement("Einwohner")).InnerText = "80767600";
```

Das Gleiche für Frankreich:

```
node = root.SelectSingleNode("EUROPA/Frankreich");
node.AppendChild(xmldoc.CreateElement("Fläche")).InnerText = "343965";
node.AppendChild(xmldoc.CreateElement("Einwohner")).InnerText = "57800000";
...
xmldoc.Save(@"c:\Test7.xml");
```

Ergebnis

Das resultierende Dokument können Sie sich im Internet Explorer ansehen:



22.3.8 Suchen in den Baumzweigen

Die hierarchischen Baumstrukturen sind zwar optisch recht anschaulich, aber programmiertechnisch ziemlich unübersichtlich. Die wohl trivialste, aber auch umständlichste Variante zum Suchen hatten wir Ihnen bereits vorgestellt. Sie können sich, wie bereits demonstriert, einfach durch die Liste der Knoten "hangeln", was jedoch sehr zeitaufwändig und wenig effizient ist. Wesentlich schneller und komfortabler geht es mit den folgenden Methoden:

Methode	Beschreibung
GetElementsByTagName(Suchstring)	Liefert eine Collection von *XMLNode*-Objekten, basierend auf dem übergebenen Suchstring bzw. Tag-Namen. Mit "*" können Sie alle Elemente abrufen. Geben Sie lediglich einen Tag-Namen an, werden Ihnen alle Vorkommen dieses Tags aufgelistet, unabhängig von der gerade aktiven Baumebene.
SelectNodes(Suchstring)	Liefert eine Liste von *XMLNode*-Objekten, die dem übergebenen Suchstring entsprechen. Der Suchstring bezieht sich im Normalfall auf die gerade aktive Baumebene.
SelectSingleNode(Suchstring)	Verhält sich wie *SelectNodes*, mit dem Unterschied, dass lediglich das erste gefundene Element zurückgegeben wird.

BEISPIEL 22.29: Verwendung *GetElementsByTagName*

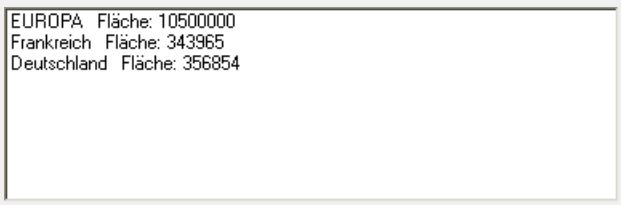
C#

Aus der Datei *test7.xml* sollen alle Elemente, unabhängig von ihrer Baumebene, ermittelt werden, die den Namen *Fläche* tragen. Die Element-Namen, der jeweilige Parent sowie der Inhalt der *Text*-Eigenschaft sollen in einer *ListBox* angezeigt werden.

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root;
XmlNodeList list;

xmldoc.Load(@"c:\test7.xml");
root = xmldoc.DocumentElement;
listBox1.Items.Clear();
list = xmldoc.GetElementsByTagName("Fläche");
for (int i = 0; i < list.Count; i++)
  listBox1.Items.Add(list[i].ParentNode.Name + "   Fläche: " + list[i].InnerText);
```

Ergebnis

Ausgabe in der *ListBox:*



Statt Sie auf den folgenden Seiten mit endlosen undurchsichtigen Auflistungen von Optionen und Parametern zu quälen, haben wir uns entschieden, Ihnen anhand einiger aussagekräftiger Beispiele die Möglichkeiten der drei genannten Methoden zu demonstrieren.

BEISPIEL 22.30: Verwenden von *SelectNodes*

C#

```
XmlDocument xmldoc = new XmlDocument();
XmlNode root;
XmlNodeList list;
xmldoc.Load(@"c:\test7.xml");
root = xmldoc.DocumentElement;

listBox1.Items.Clear();
```

Eingabe des gewünschten Suchstrings und Auswahl:

```
list = xmldoc.SelectNodes(textBox1.Text);
```

Anzeige der Daten im Listenfeld:

```
for (int i = 0; i < list.Count; i++)
{
  if (list[i].InnerText.Trim() != "")
    listBox1.Items.Add(list[i].Name + "  (Text: " + list[i].InnerText + ")");
  else
    listBox1.Items.Add(list[i].Name);
}
```

Die verschiedenen Suchstrings und das zugehörige Ergebnis:

Suchstring	Beschreibung/Ausgabe
/*	Rückgabewert ist das Root-Element: `WELT (Text: 10500000718500000543965578000003568548076760O)`
/WELT	Auswahl des Root-Elements über den Namen: `WELT (Text: 10500000718500000543965578000003568548076760O)`

/WELT/*	Anzeige aller Untereinträge von "WELT":
	`AFRIKA`
	`ANTARKTIS`
	`ASIEN`
	`AUSTRALIEN`
	`EUROPA (Text: 10500000718500000543965578000003568548076760O)`
	`NORDAMERIKA`
	`SÜDAMERIKA`

Suchstring	Beschreibung/Ausgabe

//*	Rückgabewert ist eine Collection aller Knoten:
	`WELT (Text: 10500000718500000543965578000003568548076760 0)`
	`AFRIKA`
	`ANTARKTIS`
	`ASIEN`
	`AUSTRALIEN`
	`EUROPA (Text: 10500000718500000543965578000003568548076760 0)`
	`Fläche (Text: 10500000)`
	`Einwohner (Text: 718500000)`
	`Frankreich (Text: 54396557800000)`
	`Fläche (Text: 543965)`
	`Einwohner (Text: 57800000)`
	`Deutschland (Text: 35685480767600)`
	`Fläche (Text: 356854)`
	`Einwohner (Text: 80767600)`
	`Italien`
	`Österreich`
	`Norwegen`
	`Polen`
	`NORDAMERIKA`
	`SÜDAMERIKA`

/WELT/*[not(Frankreich)]	Sucht alle Knoten unterhalb von "WELT", die keinen Unterknoten mit dem Namen "Frankreich" haben:
	`AFRIKA`
	`ANTARKTIS`
	`ASIEN`
	`AUSTRALIEN`
	`NORDAMERIKA`
	`SÜDAMERIKA`

/WELT/*[Frankreich]	Sucht alle Knoten unterhalb von "WELT", die einen Unterknoten mit dem Namen "Frankreich" haben: `EUROPA (Text: 10500000718500000543965578000003568548076760 0)`

WELT/EUROPA/*[Fläche or Einwohner]	Sucht alle Knoten unterhalb von "WELT/EUROPA", die entweder einen Unterknoten mit dem Namen "Fläche" oder "Einwohner" haben:
	`Frankreich (Text: 54396557800000)`
	`Deutschland (Text: 35685480767600)`

WELT/EUROPA/*[Fläche > 400000]	Sucht alle Knoten unterhalb von "WELT/EUROPA", die eine Fläche > 400000 haben:
	`Frankreich (Text: 54396557800000)`
	Sie können und sollten den Eintrag "Fläche" vor der Vergleichsoperation mit der Funktion *number* in einen numerischen Wert umwandeln.
	`WELT/EUROPA/*[number(Fläche) > 400000]`

/WELT/*[last()]	Sucht den letzten Knoten unterhalb von "WELT": `SÜDAMERIKA`

Suchstring	Beschreibung/Ausgabe
WELT/*[3]	Der vierte Untereintrag unter "WELT": ASIEN

Diese Übersicht dürfte Ihnen einen ersten Einblick in die Möglichkeiten der Suche gewährt haben.

22.4 XML-Verarbeitung mit LINQ to XML

Mit LINQ to XML ist eine komfortable Unterstützung für die Verarbeitung von XML-Dokumenten direkt in die jeweiligen .NET-Programmiersprachen integriert.

XML-Dokumente sind damit keine "Fremdkörper", auf die mit extra APIs und externen Objektmodellen zugegriffen werden muss, sondern der C#-Programmierer kann mit seinen gewohnten Werkzeugen direkt XML-Strukturen verarbeiten, ohne sich mit zusätzlichen Abfragesprachen (XSLT) beschäftigen zu müssen.

Die Auswertung der XML-Dokumente entspricht von der Syntax in weiten Bereichen der LINQ to Object-Syntax, Sie können also auf diese Grundlagen aufbauen, wenn Sie Daten filtern, sortieren oder transformieren wollen. Dazu benötigen Sie weder Kenntnisse von XPath, XQuery oder DOM[1]. Sie werden sehen, dass Sie in fast allen Fällen mit deutlich weniger und besser verständlichem Quellcode zum Ziel kommen.

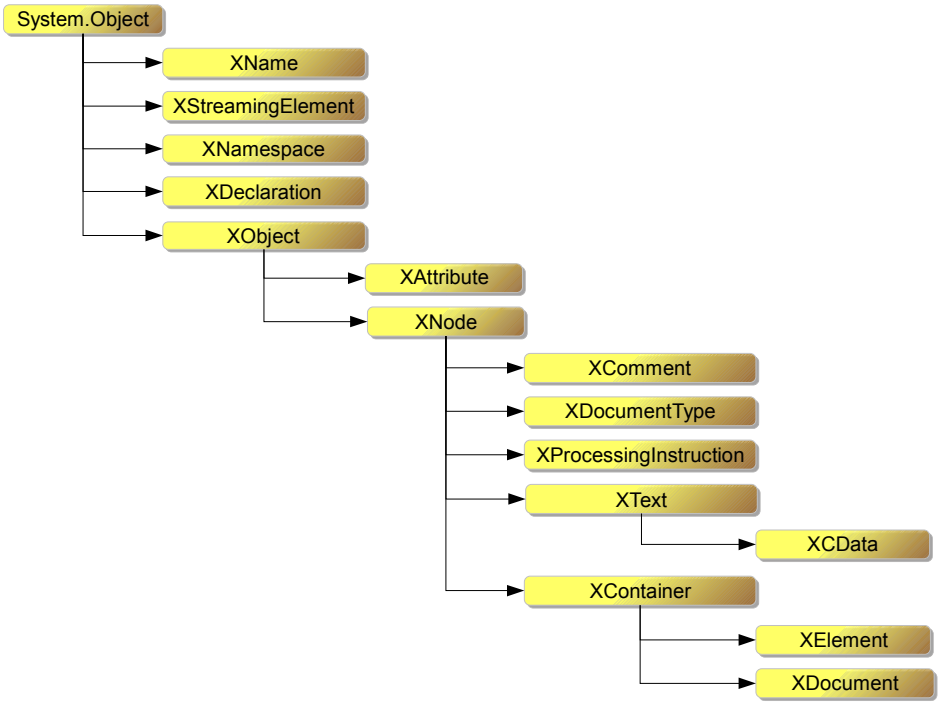
Für alle XML DOM-Entwickler (Namespace *System.Xml*), besteht kein Grund zur Sorge: Ihr bisheriger Quellcode funktioniert nach wie vor, Sie können sukzessive den Quellcode migrieren oder mit beiden Technologien arbeiten. Doch jetzt zu den Details.

22.4.1 Die LINQ to XML-API

Mit Visual Studio 2008 wurde vor einigen Jahren eine neue API für den Zugriff auf XML-Dokumente eingeführt, die unter anderem auch für die Arbeit mit LINQ to XML von Bedeutung ist. Die relevanten Klassen finden Sie im Namespace *System.Xml.Linq*, die folgende Abbildung zeigt eine Übersicht, mit den entsprechenden Abhängigkeiten.

HINWEIS: Bei den Klassen *XNode*, *XContainer* und *XObject* handelt es sich um abstrakte Klassen, Sie können diese also nicht direkt verwenden.

[1] Wobei derartige Kenntnisse niemandem geschadet haben.



In der folgenden Tabelle finden Sie eine kurze Beschreibung zu den wichtigsten Klassen der XML-API:

Klasse	Beschreibung
XDocument	Diese Klasse repräsentiert ein komplettes XML Infoset Dokument, in dem wiederum Processing Instructions (PI), XML Document Type Definitions (DTD), Kommentare und natürlich die eigentlichen Elemente mit Attributen enthalten sind. Die Klasse stellt für das Laden und Sichern statische Methoden zur Verfügung, über den Konstruktor können Sie nur neue Dokumente erzeugen bzw. ein anderes *XDocument* klonen. Da in der neuen XML-API nicht mehr unbedingt mit kompletten XML-Dokumenten gearbeitet werden muss, sollten Sie dieses Objekt nur nutzen, wenn Sie Dokumente erzeugen wollen, die über Processing Instructions und XML Document Type Definitions verfügen sollen. Andernfalls ist *XElement* die bessere Wahl.
XElement	Eine der wichtigsten neuen Klassen. *XElement* entspricht einem Element, dieses kann wiederum weitere Elemente enthalten (ein XML-Fragment). Die Klasse bietet neben Methoden zum Laden und Sichern der enthaltenen XML-Fragmente alle Möglichkeiten, Elemente/Attribute hinzuzufügen, zu

Klasse	Beschreibung
	manipulieren oder zu entfernen. Über den Konstruktor können Sie komplette XML-Baumstrukturen erzeugen, ohne auch nur eine Methode auszurufen oder explizit neue Objekte zu erzeugen. *XElement* dürfte in fast allen Fällen der Ausgangspunkt für die Arbeit mit LINQ to XML-Abfragen sein (*from*-Klausel).
XAttribute	Verwenden Sie diese Klasse zum Erzeugen/Verwalten von Attributen.
XComment	Diese Klasse repräsentiert einen XML-Kommentar.
XNode	Diese abstrakte Klasse implementiert bereits die wichtigsten Methoden zum Hinzufügen, Löschen, Ersetzen oder Verschieben von Knoten innerhalb des XML-Dokuments.
XDocumentType	Erzeugen bzw. verwalten Sie mit dieser Klasse eine DTD (*Document Type Definition*).
XProcessing-Instruction	Repräsentiert die Processing Instructions (PIs) eines XML Dokuments.
XDeclaration	Die XML-Deklaration, diese wird für die Angabe der XML-Version sowie der Kodierung (Zeichensatz, z.B. UTF-8/UTF-16) genutzt.

Nach dieser Kurzübersicht wollen wir Ihnen anhand einiger Szenarien die Verwendung der obigen Objekte demonstrieren und Sie mit den Möglichkeiten von LINQ to XML vertraut machen.

22.4.2 Neue XML-Dokumente erzeugen

LINQ to XML bietet Ihnen einen recht einfachen Weg, um selbst komplexeste XML-Dokumente zu erstellen. Im Gegensatz zum DOM müssen Sie nicht erst Instanzen für einzelne XML-Elemente erzeugen, sondern können gleich die Konstruktoren der einzelnen Elemente (*XElement*, *XAttribute* etc.) ineinander verschachteln. Möglich wird dies durch recht flexible Parameter für die jeweiligen Konstruktoren, die neben dem allgemeinen *Object*-Datentyp auch optionale Parameterlisten entgegennehmen können.

BEISPIEL 22.31: Neues Dokument erzeugen

C#

Sie möchten folgendes XML-Dokument erzeugen:

```
<Artikelliste>
 <Artikel artid="1">
   <ArtikelNr>1</ArtikelNr>
   <Artikelname>Chai</Artikelname>
   <Einzelpreis>18</Einzelpreis>
   <Lagerbestand>18</Lagerbestand>
 </Artikel>
 <Artikel artid="2">
   <ArtikelNr>2</ArtikelNr>
   <Artikelname>Chang</Artikelname>
```

BEISPIEL 22.31: Neues Dokument erzeugen

```
    <Einzelpreis>19</Einzelpreis>
    <Lagerbestand>17</Lagerbestand>
  </Artikel>
</Artikelliste>
```

Die Umsetzung mit Hilfe geschachtelter Konstruktoren (funktionale Konstruktion):

```
...
  private XElement ArtikelListe;
...
          ArtikelListe = new XElement("Artikelliste",
                               new XElement("Artikel", new XAttribute("artid",1),
                                   new XElement("ArtikelNr", 1),
                                   new XElement("Artikelname", "Chai"),
                                   new XElement("Einzelpreis", 18),
                                   new XElement("Lagerbestand", 18)
                               ),
                               new XElement("Artikel", new XAttribute("artid",2),
                                   new XElement("ArtikelNr", 2),
                                   new XElement("Artikelname", "Chang"),
                                   new XElement("Einzelpreis", 19),
                                   new XElement("Lagerbestand", 17)
                               )
                        );
...
```

Die Schachtelung der einzelnen Objekt-Konstruktoren orientiert sich hier an der Schachtelung der XML-Tags (inklusive der Definition von Attributen). Im Gegensatz zu einem *XmlWriter*, mit dem eine ähnliche kurze Schreibweise möglich ist, brauchen Sie sich in diesem Fall nicht um das Schließen der jeweiligen Tags zu kümmern, dies resultiert bereits aus obiger Klammersetzung. Fehlerhafte End-Tags gehören damit der Vergangenheit an, die richtige Klammersetzung wird bereits zur Entwurfszeit kontrolliert. Zusätzlich werden auch die Namen der Knoten geprüft (XML-Namensregeln), sowie die Inhalte gegebenenfalls kodiert (Sonderzeichen Escaping).

Wie schon einleitend erwähnt, können Sie über ein *XDocument*-Objekt auch Kommentare, Processing Instructions und Deklarationen erstellen, um ein komplettes XML-Dokument zu erzeugen.

BEISPIEL 22.32: Deklaration und Kommentar in ein *XDocument*-Objekt einfügen

C#

```
XDocument mydoc = new XDocument(
                      new XDeclaration("1.0", "utf-8", "yes"),
                      new XComment("Artikeldatenbank"),
                      new XElement("Artikelliste",
                      ...
```

22.4.3 Laden und Sichern von XML-Dokumenten

Handelt es sich um vorhandene XML-Daten, müssen diese zunächst in ein *XDocument*- oder *XElement*-Objekt geladen werden, bevor eine Verarbeitung möglich ist. Dazu bietet sich die naheliegende statische Methode *Load* an, die sich erfreulicherweise auch recht "kontaktfreudig" gibt. So können Sie die Daten entweder aus einer Datei (auch URL), einem *XmlReader* oder einem *TextReader* einlesen:

```
public static XElement Load(string uri);
public static XElement Load(string uri, LoadOptions options);
public static XElement Load(TextReader textReader);
public static XElement Load(TextReader textReader, LoadOptions options);
public static XElement Load(XmlReader reader);
public static XElement Load(XmlReader reader, LoadOptions options)
public static XElement Parse(string text);
public static XElement Parse(string text, LoadOptions options);
```

Alternativ bietet sich die *Parse*-Methode an, um die Daten aus einer Zeichenkette einzulesen.

BEISPIEL 22.33: XML-Inhalte aus einer Datei laden

C#
```
XElement ArtikelListe = XElement.Load("Artikel.xml");
```

BEISPIEL 22.34: Laden aus einer Zeichenkette

C#
```
XElement ArtikelListe = XElement.Parse("<Artikelliste><Artikel><Name>Junger Hund</Name>
                                        <Preis>120.17</Preis></Artikel></Artikelliste>");
MessageBox.Show(ArtikelListe.ToString(), "Artikelliste");
```

Ergebnis

Die angezeigten Daten:

Nach dem Einlesen der XML-Daten können Sie diese filtern, löschen, sortieren, konvertieren oder editieren, um sie abschließend wieder in eine persistente Form zu bringen. Hier bieten sich die verschiedenen Überladungen der *Save*-Methode zur Ausgabe in eine Datei, einen *TextWriter* oder einen *XmlWriter* an:

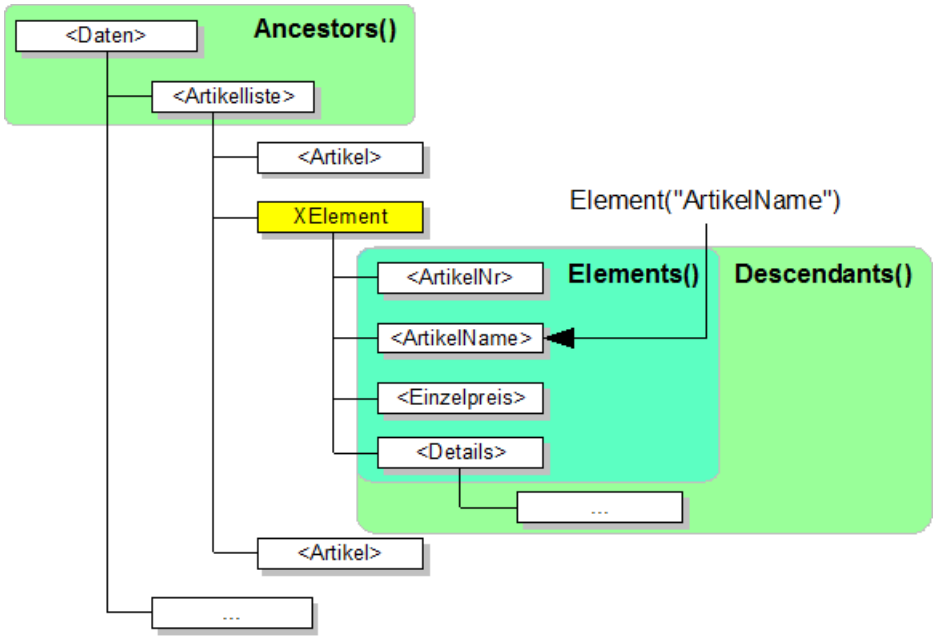
```
public void Save(string fileName);
public void Save(TextWriter textWriter);
public void Save(XmlWriter writer);
public void Save(string fileName, SaveOptions options);
public void Save(TextWriter textWriter, SaveOptions options);
```

BEISPIEL 22.35: Sichern in einer XML-Datei

```
ArtikelListe.Save("ArtikelNeu.xml");
```

22.4.4 Navigieren in XML-Daten

Wie schon erwähnt, sind *XElement* oder *XDocument* die geeigneten Klassen für die Arbeit mit den darin abgebildeten XML-Daten. Beide Klassen bieten eine Reihe von Methoden (geerbt von *XContainer*/*XNode*), mit denen Sie auf die enthaltenen XML-Knoten zugreifen bzw. zwischen diesen navigieren können (siehe folgende Abbildung).



Die folgende Tabelle zeigt weitere Methoden und deren Beschreibung:

Methode	Beschreibung
Ancestors	Rekursive *XElement*-Liste aller übergeordneten Elemente des aktuellen Knotens.
CompareDocument-Order	Der relative Abstand zweier Knoten (*XNode*).

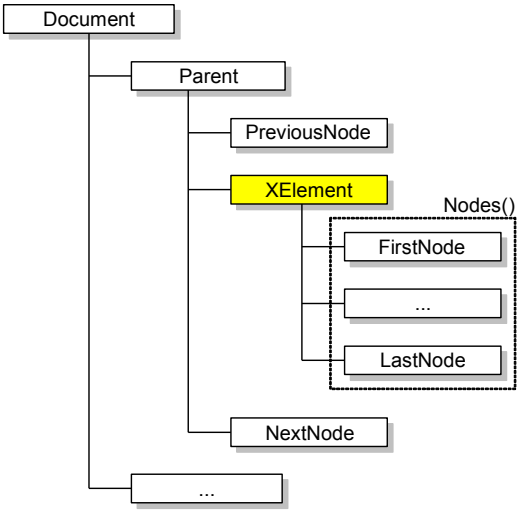
Methode	Beschreibung
Descendants	Rekursive *XElement*-Liste aller untergeordneten Elemente des aktuellen Knotens.
Element	Auswahl eines untergeordneten Elements über dessen Namen.
Elements	Auswahl der untergeordneten Elemente (alternativ mit Angabe des Namens). Zurückgegeben wird eine *IEnumerable* Liste.
IsAfter	Liefert *True*, wenn sich der aktuelle Knoten hinter dem Vergleichsknoten befindet.
IsBefore	Liefert *True*, wenn der aktuelle Knoten sich vor dem Vergleichsknoten befindet.
Nodes	Liefert eine Liste der untergeordneten Knoten.

BEISPIEL 22.36: Anzeige aller Artikelnamen in einem Listenfeld

C#
```
XElement ArtikelListe = XElement.Load("Artikel.xml");
foreach(XElement Artikel in ArtikelListe.Elements())
   ListBox1.Items.Add(Artikel.Element("Artikelname").Value);
```

Nach dem Laden des Dokuments ist der aktuelle Knoten "Artikelliste". Mit *Elements* rufen wir alle "Artikel"-Knoten ab, über das spezifische Unterelement "Artikelname" kommen wir schließlich an den gewünschten Wert.

Neben den oben genannten Methoden stehen auch einige Eigenschaften für die Navigation zwischen den Baumknoten zur Verfügung.



So bieten *FirstNode*, *NextNode*, *LastNode*, *PreviousNode* und *Parent* jeweils Referenzen auf die umliegenden Knoten/Elemente. Wir gehen an dieser Stelle nicht weiter auf diese Eigenschaften ein, da Sie Ihnen sicher bereits von XML DOM-Anwendungen her bekannt sind.

Wesentlich interessanter ist noch ein Blick auf die Möglichkeiten, die sich mit den Erweiterungsmethoden bieten. Da viele der obigen Methoden Listen zurückgeben, können wir auch die dazu passenden Methoden (z.B. *First*, *Last*, *Skip*, *Take* ...) verwenden.

BEISPIEL 22.37: Neue Liste mit 20 Artikeln bilden, ohne die ersten 5 Einträge

C#
```
var Artikelliste2 = ArtikelListe.Elements().Skip(5).Take(20);
```

22.4.5 Auswählen und Filtern

Einige einfache Varianten zum Filtern der XML-Daten haben Sie bereits im vorhergehenden Abschnitt kennen gelernt. So können Sie mit *Take* eine vorgegebene Anzahl von Datensätzen abrufen bzw. mit *Skip* eine bestimmte Anzahl von Elementen überspringen. Beide Varianten gibt es auch mit einer Bedingung (*SkipWhile*, *TakeWhile*).

Dem SQL-Programmierer dürfte beim Blick in die Liste der möglichen Methoden die *Where*-Methode sicher nicht entgangen sein. Hier haben wir es mit einer echten Filterbedingung zu tun, die auf das Element bzw. die enthaltenen Elemente und Attribute zurückgreifen kann.

BEISPIEL 22.38: Filtern der Artikelliste (Einzelpreis < 20)

C#
```
XElement ArtikelListe
ArtikelListe = XElement.Load("Artikel.xml");
var Artikelliste2 = ArtikelListe.Elements().Where(a => (float)a.Element("Einzelpreis") < 20);
```

Mit einem einfachen Lambda-Ausdruck ist die gewünschte Aufgabenstellung problemlos realisierbar. Wer es gern etwas übersichtlicher hätte, kann auch die folgende Schreibweise verwenden:

BEISPIEL 22.39: Filtern der Artikelliste (Einzelpreis < 20)

C#
```
var Artikelliste3 = from a in ArtikelListe.Elements()
                    where (float)a.Element("Einzelpreis") < 20
                    select a;
```

Auf die umfangreichen Möglichkeiten, Filterbedingungen zusammenzustellen, wollen wir an dieser Stelle nicht weiter eingehen, dazu wurden bereits im LINQ-Kapitel (Abschnitt 6.3) die Grundlagen gelegt.

22.4.6 Manipulieren der XML-Daten

Nachdem die XML-Daten geladen und gefiltert sind, wird sicher bald der Wunsch aufkommen, diese Daten auch zu bearbeiten. Dazu zählen wir das Editieren, Löschen und Hinzufügen neuer Inhalte.

Editieren der Daten

Das Editieren von Elementen oder Attributen stellt sich als relativ simpler Vorgang dar. Sie navigieren mit Hilfe der *XElement*-Methoden zum gewünschten Knoten und können hier mit den Methoden *SetElementValue* bzw. *SetAttributeValue* die gewünschten Änderungen vornehmen.

BEISPIEL 22.40: Editieren eines Elements

C#
```
var Artikel = ArtikelListe.Elements().Skip(1).First();
MessageBox.Show(Artikel.ToString(), "XML");
Artikel.SetElementValue("Einzelpreis", 200);
MessageBox.Show(Artikel.ToString(), "XML");
```

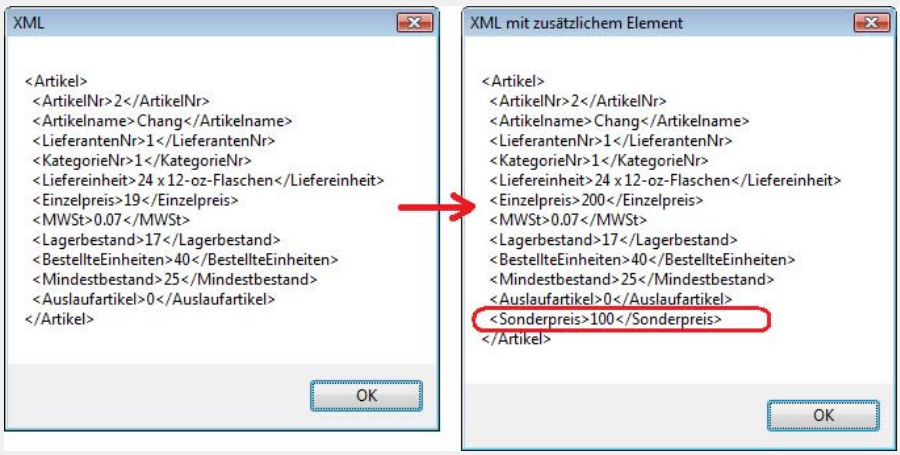
Hinzufügen von Daten

Mit den Methoden *Add*, *AddFirst*, *AddBeforeSelf*, *AddAfterSelf* können Sie neue Elemente hinzufügen.

BEISPIEL 22.41: Hinzufügen eines untergeordneten Elements

C#
```
var Artikel = ArtikelListe.Elements().Skip(1).First();
Artikel.Add(new XElement("Sonderpreis", "100"));
```

Ergebnis



BEISPIEL 22.42: Hinzufügen eines kompletten XML-Fragments (Artikel) in die Artikelliste

C#
```
ArtikelListe.Add(new XElement("Artikel",
                    new XElement("ArtikelNr", 2317),
                    new XElement("Artikelname", "Junger Hund"),
                    new XElement("LieferantenNr", 1),
                    new XElement("KategorieNr", 1),
                    new XElement("Liefereinheit", "10 Kartons x 20 Dosen"),
                    new XElement("Einzelpreis", 18),
                    new XElement("MWSt", 0.07),
```

BEISPIEL 22.42: Hinzufügen eines kompletten XML-Fragments (Artikel) in die Artikelliste

```
                    new XElement("Lagerbestand", 18),
                    new XElement("BestellteEinheiten", 0),
                    new XElement("Mindestbestand", 10),
                    new XElement("Auslaufartikel", 0)));
```

Löschen

Auch das Löschen von Inhalten stellt keine große Hürde dar. Die *XElement*-Klasse bietet dazu eine Reihe geeigneter Methoden:

Methode	Beschreibung
Remove	Löscht den aktuellen Knoten.
RemoveAll	Löscht alle untergeordneten Elemente und Attribute.
RemoveAnnotations	Löscht Anmerkung des spezifizierten Typs.
RemoveAttributes	Löscht alle Attribute des aktuellen Knotens.
RemoveContent	Löscht alle untergeordneten Elemente, aber nicht die Attribute.

22.4.7 XML-Dokumente transformieren

Neben der reinen Verarbeitung von XML-Daten ist es häufig auch erforderlich, diese Daten in eine andere Form zu bringen. Dem professionellen XML-Programmierer fällt in diesem Zusammenhang sicher gleich XSLT ein, doch nicht jeder möchte sich gleich in eine andere und dazu noch recht komplexe Sprache einarbeiten. Viel schöner wäre es, wenn Sie diese Aufgabe gleich im Quelltext Ihres C#-Programms realisieren könnten.

LINQ to XML ist genau die richtige Lösung für dieses Szenario. Mit wenigen Zeilen Code können Sie XML-Daten aus beliebigen Quellen laden, bearbeiten und nachfolgend in neuer Form sichern bzw. zurückgeben. Der interessanteste Ansatzpunkt ist in diesem Zusammenhang die *select*-Klausel, mit der Sie die Daten in eine neue Form projizieren können.

An drei einfachen Beispielen demonstrieren wir Ihnen die Vorgehensweise:

- Filtern der Daten, Berechnung eines neuen Feldes und Projektion der XML-Elemente in XML-Attribute,
- Konvertieren in ein HTML-Dokument mit Tabellen-Layout,
- Konvertieren in eine Objektliste, die Sie in einem *DataGridView* anzeigen können.

Sie werden sehen, dass der grundlegende Ansatz in allen drei Beispielen gleich ist:

- Zunächst wird ein bereits gefülltes *XElement* abgefragt (*from*),
- nachfolgend werden die Filterbedingungen definiert (*where*),
- abschließend werden die Daten in die gewünschte Form gebracht (*select*).

BEISPIEL 22.43: Eine gefüllte XML-Artikelliste soll gefiltert und der Wert (Anzahl * Einzelpreis) der einzelnen Waren berechnet werden. Abschließend werden diese Daten in einer kompakten Form (Attribute statt Elemente) als XML-Datei ausgegeben.

C#

```
XElement ArtikelListe;
ArtikelListe = XElement.Load("Artikel.xml");
XElement Abfrage = new XElement("Artikel",
                       from artikel in ArtikelListe.Elements()
                       where (int)artikel.Element("ArtikelNr") > 5
                       where (int)artikel.Element("ArtikelNr") < 20
                       select new XElement("Art",
                           new XAttribute("Nr", artikel.Element("ArtikelNr").Value),
                           new XAttribute("Name", artikel.Element("Artikelname").Value),
                           new XAttribute("Wert", (float)artikel.Element("Lagerbestand") *
                                                  (float)artikel.Element("Einzelpreis"))
                                      )
                                  );
Abfrage.Save("gefiltert.xml");
```

Ergebnis

```
<?xml version="1.0" encoding="utf-8" ?>
- <Artikel>
    <Art Nr="6" Name="Grandma's Boysenberry Spread" Wert="3000" />
    <Art Nr="7" Name="Uncle Bob's Organic Dried Pears" Wert="450" />
    <Art Nr="8" Name="Northwoods Cranberry Sauce" Wert="240" />
    <Art Nr="9" Name="Mishi Kobe Niku" Wert="2813" />
    <Art Nr="10" Name="Ikura" Wert="961" />
    <Art Nr="11" Name="Queso Cabrales" Wert="462" />
    <Art Nr="12" Name="Queso Manchego La Pastora" Wert="3268" />
    <Art Nr="13" Name="Konbu" Wert="144" />
    <Art Nr="14" Name="Tofu" Wert="813.75" />
    <Art Nr="15" Name="Genen Shouyu" Wert="604.5" />
    <Art Nr="16" Name="Pavlova" Wert="506.050018" />
    <Art Nr="17" Name="Alice Mutton" Wert="0" />
    <Art Nr="18" Name="Carnarvon Tigers" Wert="2625" />
    <Art Nr="19" Name="Teatime Chocolate Biscuits" Wert="230" />
  </Artikel>
```

BEISPIEL 22.44: Wie im obigen Beispiel sollen die Daten abgefragt, jetzt jedoch im HTML-Format gesichert werden.

C#

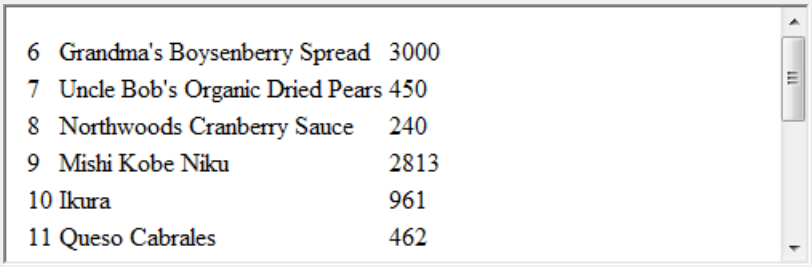
```
...
XElement html =  new XElement("html", new XElement("body", new XElement("table",
                     from artikel in ArtikelListe.Elements()
                     where (int)artikel.Element("ArtikelNr") > 5
                     where (int)artikel.Element("ArtikelNr") < 20
                     select new XElement("tr",
                        new XElement("td", artikel.Element("ArtikelNr").Value),
                        new XElement("td", artikel.Element("Artikelname").Value),
                        new XElement("td", (float)artikel.Element("Lagerbestand") *
                                           (float)artikel.Element("Einzelpreis"))
                     )
```

BEISPIEL 22.44: Wie im obigen Beispiel sollen die Daten abgefragt, jetzt jedoch im HTML-Format gesichert werden.

```
                ));
html.Save("artikel.html");
```

Ergebnis

Die ausgegebene Datei:



BEISPIEL 22.45: Konvertieren in eine typisierte Objektliste zur Anzeige in einer *DataGridView*

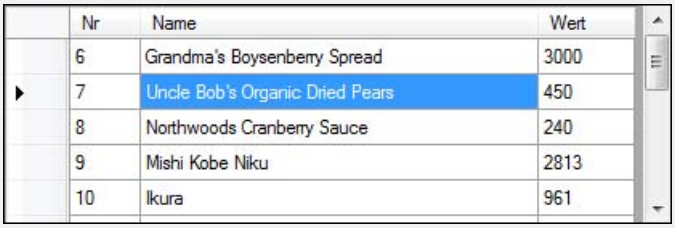
C#

```
...
var Abfrage = from artikel in ArtikelListe.Elements()
              where (int)artikel.Element("ArtikelNr") > 5
              where (int)artikel.Element("ArtikelNr") < 20
              select new { Nr = (int)artikel.Element("ArtikelNr"),
                           Name = (string)artikel.Element("Artikelname"),
                           Wert = ((float)artikel.Element("Lagerbestand") *
                                  (float)artikel.Element("Einzelpreis"))};
dataGridView1.DataSource = Abfrage.ToList();
```

In diesem Fall wird mittels *select* ein neues Objekt (anonymer Typ) erzeugt, das über die drei Eigenschaften *Nr*, *Name* und *Wert* verfügt.

Ergebnis

Die Daten im *DataGridView*:



Nr	Name	Wert
6	Grandma's Boysenberry Spread	3000
7	Uncle Bob's Organic Dried Pears	450
8	Northwoods Cranberry Sauce	240
9	Mishi Kobe Niku	2813
10	Ikura	961

HINWEIS: Vergessen Sie nicht die Umwandlung in eine *List*, andernfalls bleibt das *DataGridView* leer.

Damit dürfte die Leistungsfähigkeit von LINQ to XML erkennbar geworden sein und dem Datenaustausch mit anderen bekannten Datenkonstrukten (DataSet, Array etc.) nichts mehr im Wege stehen.

22.5 Weitere Möglichkeiten der XML-Verarbeitung

XML ist unter .NET allgegenwärtig. Wir geben hier lediglich eine kleine Auswahl der Einsatzmöglichkeiten.

22.5.1 Die relationale Sicht mit XmlDataDocument

Ausgehend von den Ausführungen des Abschnitts 22.3 wird man feststellen, dass sich mit dem DOM zwar hierarchische XML-Dateien gut verarbeiten lassen, der Zusammenhang zur relationalen Welt der Datenbanken fehlt jedoch. Gerade das .NET Framework bzw. das enthaltene ADO.-NET machen von XML reichlich Gebrauch. Was liegt also näher, als beide Welten miteinander zu verschmelzen?

Die Rede ist vom *XmlDataDocument*-Objekt, das sowohl relationale als auch XML-Daten laden, verarbeiten und speichern kann. *XmlDataDocument* ist von der Klasse *XmlDocument* abgeleitet, verfügt also über alle erforderlichen Eigenschaften um mit dem DOM zu arbeiten. Gleichzeitig findet sich jedoch auch eine neue Eigenschaft *DataSet*. Genau diese stellt für uns die Verbindung zu den relationalen Daten her.

Laden von XML-Daten

Laden können Sie reine XML-Daten entweder über das *DataSet* oder auch über das *XMLDataDocument*-Objekt.

BEISPIEL 22.46: Laden der Daten über das DataSet

C#

```
DataSet ds = new DataSet();
ds.ReadXml(@"c:\test7.xml");
```

Hier wird das *XMLDataDocument* mit dem *DataSet* synchronisiert:

```
XmlDataDocument xmlddoc = new XmlDataDocument(ds);
```

Alternativ können Sie jedoch auch erstmal ein XSD-Schema erzeugen:

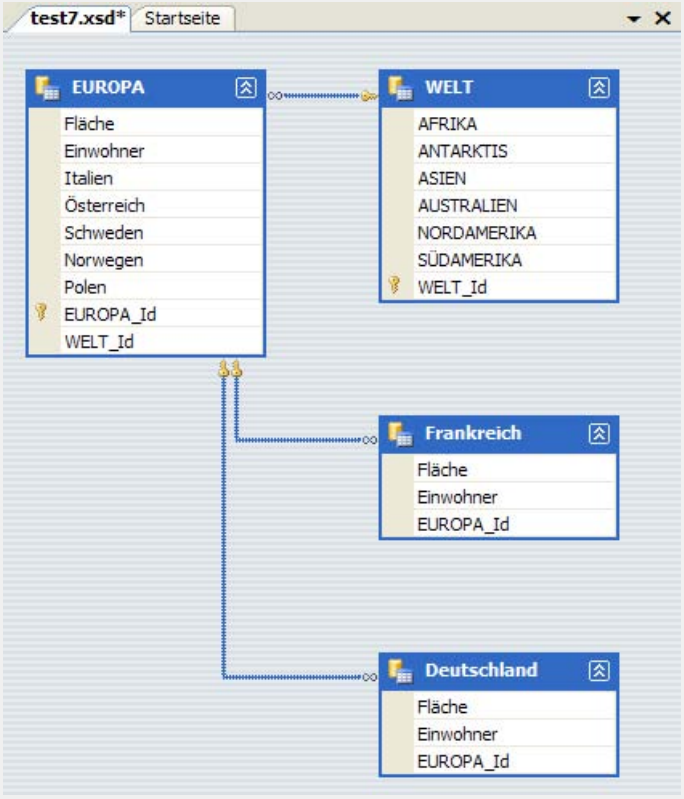
```
DataSet ds = new DataSet();

ds.ReadXml(@"c:\test7.xml");
ds.WriteXmlSchema(@"c:\test7.xsd");
```

Warum dieser zusätzliche Aufwand? Die Antwort ist schnell gegeben wenn Sie versuchen, über das *XMLDataDocument*-Objekt eine XML-Datei zu laden, ohne entsprechende Schema-Informationen bereitzustellen. Der Versuch wird zwar erfolgreich sein, im *DataSet* werden Sie jedoch keine Tabelle vorfinden.

BEISPIEL 22.46: Laden der Daten über das DataSet

Ergebnis

Das erzeugte Schema für unsere XML-Datei:



BEISPIEL 22.47: Laden des Schemas und nachfolgendes Laden der XML-Daten

C#

```
XmlDataDocument xmlddoc = new XmlDataDocument();
xmlddoc.DataSet.ReadXmlSchema(@"c:\test7.xsd");
xmlddoc.Load(@"c:\test7.xml");
```

Verwendung des DataSets

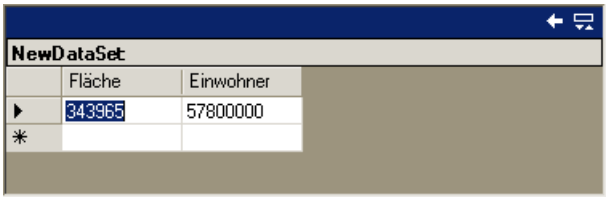
Einer Anzeige, zum Beispiel in einem *DataGrid*, steht jetzt nichts mehr im Wege:

```
XmlDataDocument xmlddoc = new XmlDataDocument();
xmlddoc.DataSet.ReadXmlSchema(@"c:\test7.xsd");
xmlddoc.Load(@"c:\test7.xml");
dataGrid1.DataSource = xmlddoc.DataSet;
```

HINWEIS: Wir verwenden in diesem Fall kein *DataGridView*, sondern das "gute alte" *DataGrid*, da dieses zur Laufzeit eine Auswahl der enthaltenen *DataTable*s anbietet.

Ansicht des *DataSet*s im *DataGrid:*

Detailansicht (Frankreich) im *DataGrid:*



Importieren Sie die Schema-Datei in Ihr Visual Studio-Projekt, so wird automatisch eine entsprechende Designer-Datei erzeugt, die das zugehörige *DataSet* als Klassendefinition beschreibt.

BEISPIEL 22.48: Ausschnitt aus *Test7.Designer.cs*

C#

```
//------------------------------------------------------------------------------
// <auto-generated>
//     Dieser Code wurde von einem Tool generiert.
//     Laufzeitversion:2.0.50727.42
//     Änderungen an dieser Datei können falsches Verhalten verursachen und gehen verloren,
//     wenn der Code erneut generiert wird.
// </auto-generated>
//------------------------------------------------------------------------------

#pragma warning disable 1591
namespace DOM_Beispiel {
    using System;
...
    public partial class NewDataSet : System.Data.DataSet {
        private WELTDataTable tableWELT;
        private EUROPADataTable tableEUROPA;
        private FrankreichDataTable tableFrankreich;
        private DeutschlandDataTable tableDeutschland;
        private System.Data.DataRelation relationWELT_EUROPA;
        private System.Data.DataRelation relationEUROPA_Frankreich;
        private System.Data.DataRelation relationEUROPA_Deutschland;
...
    }
}
```

22.5.2 XML-Daten aus Objektstrukturen erzeugen

Nicht in jedem Fall befinden sich Ihre Ausgangsdaten für einen XML-Export schon in einem geeigneten Format (*DataTable* etc.). Doch ganz so weit brauchen Sie eigentlich gar nicht zu gehen. Wer den Pfad der reinen objektorientierten Programmierung nicht allzu weit verlassen hat, der kommt auf ganz unspektakuläre Weise zu seinen XML-Daten. Das Stichwort lautet in diesem Fall "Serialisierung" bzw. "Serialisierung im XML-Format". C# unterstützt Sie bei dieser Aufgabe mit dem *XmlSerializer*.

Einige Einschränkungen sind allerdings zu beachten:

- Die Objekte (bzw. die zugrunde liegende Klasse) muss einen Standard-Konstruktor aufweisen (ohne Parameter).
- Nur öffentliche Eigenschaften und Felder lassen sich exportieren.
- Die Eigenschaften müssen einen Schreib-/Lesezugriff ermöglichen.
- Soll eine Collection von Objekten exportiert werden, muss die Klasse von *System.Collections.*CollectionBase abgeleitet werden. Alternativ können Sie auch ein streng typisiertes Array verwenden.

BEISPIEL 22.49: Exportieren eines einfachen Objektarrays im XML-Format

C#

Definieren des Objekts (Eigenschaften *Bezeichnung*, *Anzahl*, *Preis*):

```
using System;
namespace WindowsApplication1
{
    public class Artikel
    {
        private string _Bezeichnung;

        public string Bezeichnung
        {
            get { return _Bezeichnung; }
            set { _Bezeichnung = value; }
        }

        private int _Anzahl;

        public int Anzahl
        {
            get { return _Anzahl; }
            set { _Anzahl = value; }
        }

        public Single Preis;
    }
}
```

BEISPIEL 22.49: Exportieren eines einfachen Objektarrays im XML-Format

Erzeugen eines Objektarrays:

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;

        Artikel[] Artikelliste = new Artikel[4];
...
            for (int i = 0; i < 4; i++) Artikelliste[i] = new Artikel();
            Artikelliste[0].Anzahl = 10;
            Artikelliste[0].Bezeichnung = "Mülleimer";
            Artikelliste[0].Preis = 123.45f;
            Artikelliste[1].Anzahl = 245;
            Artikelliste[1].Bezeichnung = "Osterhasen";
            Artikelliste[1].Preis = 0.99f;
            Artikelliste[2].Anzahl = 44;
            Artikelliste[2].Bezeichnung = "Schuhe";
            Artikelliste[2].Preis = 68.33f;
            Artikelliste[3].Anzahl = 2;
            Artikelliste[3].Bezeichnung = "Hosen";
            Artikelliste[3].Preis = 13.45f;
```

Speichern im XML-Format:

```
            XmlSerializer seria = new XmlSerializer(typeof(Artikel[]));
            FileStream fs = new FileStream(@"c:\MeineArtikel.xml", FileMode.Create);
            seria.Serialize(fs, Artikelliste);
            fs.Close();
```

Ergebnis

Die erzeugte Datei *MeineArtikel.xml* können Sie sich im Internet Explorer ansehen:

Das sieht doch schon ganz gut aus, allerdings haben wir bisher noch keinen Einfluss auf die Gestaltung der XML-Daten genommen. So werden die Eigenschaften pauschal als Elemente gespeichert, die Root wird automatisch benannt und auch die Elementnamen entsprechen zunächst den jeweiligen Eigenschaftsnamen.

Wem die obige Ausgabe nicht genügt, dem bietet sich die Möglichkeit, mit Hilfe von Attributen in der Objektdefinition das spätere XML-Format zu beeinflussen.

Attribute für die XML-Serialisierung:

Attribut	Beschreibung
XmlRoot	... wird der Klasse zugeordnet, es bestimmt den Namen des Root-Elements.
XmlElement	... weist der Eigenschaft/dem Member einen alternativen Elementnamen zu.
XmlAttribute	... bestimmt, das Eigenschaften als XML-Attribute statt als Elemente gespeichert werden.
XmlEnum	... bestimmt einen alternativen Bezeichner für Aufzählungen.
XmlIgnore	... die markierte Eigenschaft wird nicht in die XML-Daten aufgenommen.

HINWEIS: Die Attribute werden vom Namenspace *System.Xml.Serialization* bereitgestellt, Sie müssen diesen also auch bei der Klassendefinition einbinden.

BEISPIEL 22.50: Eine paar Änderungen an unserer Klassendefinition zeigen die Auswirkung obiger Attribute

C#

```
using System;
using System.Xml.Serialization;

namespace WindowsApplication1
{

    public class Artikel
    {
        private string _Bezeichnung;
```

Eine neue Bezeichnung festlegen und als Attribut speichern:

```
        [XmlAttribute("Name")]
        public string Bezeichnung
        {
            get { return _Bezeichnung; }
            set { _Bezeichnung = value; }
        }

        private int _Anzahl;
```

Eine neue Bezeichnung festlegen und als Attribut speichern:

```
        [XmlAttribute("Menge")]
```

BEISPIEL 22.50: Eine paar Änderungen an unserer Klassendefinition zeigen die Auswirkung obiger Attribute

```
        public int Anzahl
        {
            get { return _Anzahl; }
            set { _Anzahl = value; }
        }
```

Nicht in die XML-Daten mit aufnehmen:

```
        [XmlIgnore]
        public Single Preis;
    }
```

Eine übergeordnete Klasse definieren, so haben wir auch Einfluss auf den Namen der Root:

```
    [XmlRoot("Warenbestand")]
    public class ArtikelListe
    {
        [XmlElement("Artikel")]
        public Artikel[] art;
    }
}
```

Ergebnis

Speichern Sie ein Objekt vom Typ *ArtikelListe* ab, so erhalten Sie die folgende XML-Ausgabe:

```
<?xml version="1.0" ?>
- <Warenbestand xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <Artikel Name="Mülleimer" Menge="10" />
    <Artikel Name="Osterhasen" Menge="245" />
    <Artikel Name="Schuhe" Menge="44" />
    <Artikel Name="Hosen" Menge="2" />
  </Warenbestand>
```

22.5.3 Schnelles Suchen in XML-Daten mit XPathNavigator

Auch wenn wir im Abschnitt über das DOM bereits mit der Suche beschäftigt waren, wollen wir noch einmal auf dieses Thema eingehen. Der Grund dafür ist leicht gefunden wenn Sie versuchen, mit Hilfe des XML-DOM größere Datenmengen zu verarbeiten.

Als schnelle Alternative bietet sich der *XPathNavigator* an, den Sie mit der Methode *CreateNavigator* entweder aus einem *XPathDocument* oder einem *XmlDocument*-Objekt erzeugen können.

HINWEIS: Verwenden Sie ein *XPathDocument* zum Erzeugen, dann ist die Datenbasis schreibgeschützt.

Neben den bereits bekannten Navigationsmethoden (*MoveToNext*, *MoveToPrevious, MoveToParent* ...) dürfte vor allem die *Select*-Methode von Interesse sein. Dieser übergeben Sie einen XPath-Ausdruck, der Rückgabewert ist ein *XPathNodeIterator*, mit dem Sie die ausgewählten Knoten durchlaufen können.

Ein Beispiel dürfte für mehr Klarheit sorgen:

BEISPIEL 22.51: Auswahl aller Orte in einer XML-Datei und Ausgabe in einer *ListBox*

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;
```

Die erforderlichen Namespaces einbinden:

```
using System.Xml;
using System.Xml.XPath;
...
    public partial class Form1 : Form
    {
```

Dokument laden:

```
        private XPathDocument xdoc = new XPathDocument("telefon.xml");
        private XPathNavigator xnav;

        private void Form1_Load(object sender, EventArgs e)
        {
```

Der *XPathNavigator* wird über ein *XPathDocument* erzeugt:

```
            xnav = xdoc.CreateNavigator();
            listBox1.Items.Clear();
            try
            {
```

Daten selektieren:

```
                XPathNodeIterator xit = xnav.Select(textBox1.Text);
                label1.Text = "Fundstellen: " + xit.Count.ToString();
```

Auslesen der einzelnen Elemente und Anzeige in der *ListBox*:

```
                while (xit.MoveNext())
                {
                    listBox1.Items.Add(xit.Current.Value.ToString());
                }
            }
```

BEISPIEL 22.51: Auswahl aller Orte in einer XML-Datei und Ausgabe in einer *ListBox*

Falls ein fehlerhafter XPath-Ausdruck eingegeben wurde:

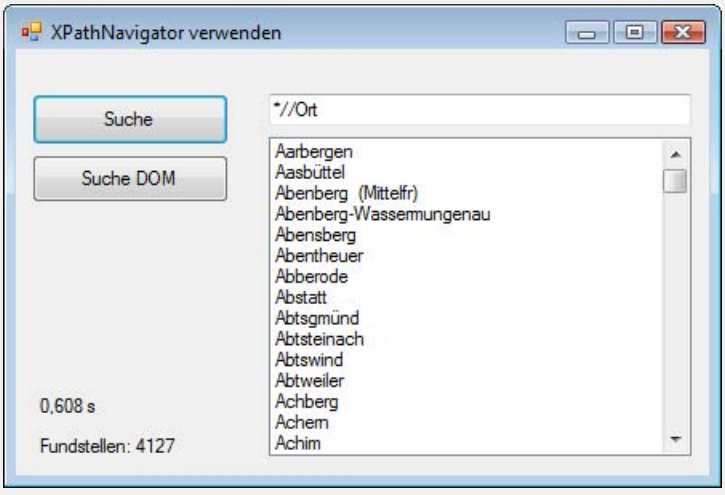
```
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
```

Ergebnis

Die Struktur der zu ladenden XML-Datei (*Telefon.xml*):

```
<?xml version="1.0" encoding="UTF-8" ?>
- <dataroot xmlns:od="urn:schemas-microsoft-com:officedata"
    generated="2006-03-08T10:59:06">
  - <Telefon>
      <Ort>Aarbergen</Ort>
      <Vorwahl>06120</Vorwahl>
    </Telefon>
  - <Telefon>
      <Ort>Aasbüttel</Ort>
      <Vorwahl>04892</Vorwahl>
    </Telefon>
  - <Telefon>
      <Ort>Abenberg (Mittelfr)</Ort>
      <Vorwahl>09178</Vorwahl>
    </Telefon>
  - <Telefon>
      <Ort>Abenberg-Wassermungenau</Ort>
```

Das Beispielprogramm in Aktion:



HINWEIS: Möchten Sie die Telefonnummern auslesen, können Sie den XPath-Ausdruck "*//Vorwahl" verwenden.

22.5.4 Schnelles Auslesen von XML-Daten mit XmlReader

Für den schnellen Lesezugriff auf XML-Dokumente bietet sich die Verwendung eines *XmlReader*s an. Dieser unterliegt jedoch gewissen Einschränkungen, die es zu beachten gilt, bevor man sich für die Verwendung entscheidet:

- nur Lesezugriff,
- nur Vorwärtsbewegung möglich,
- kein Caching.

Wer damit leben kann, der wird mit einer schnellen und speicherschonenden Alternative belohnt.

BEISPIEL 22.52: Einlesen einer XML-Datei mit Telefondaten (Struktur siehe Abbildung) in eine *ListBox*. Es sollen nur die Elemente der dritten Ebene (der Name des Ortes und die Vorwahl) angezeigt werden.

C#

```
using System.Xml;
using System.Xml.XPath;
...
    public partial class Form1 : Form
    {
        private void button1_Click(object sender, EventArgs e)
        {
            listBox1.Items.Clear();
```

Einstellungen für den *XmlReader* definieren:

```
            XmlReaderSettings myset = new XmlReaderSettings();
```

Leerzeichen ignorieren (fast immer angebracht):

```
            myset.IgnoreWhitespace = true;
```

XmlReader erzeugen (die Daten werden **nicht** in den Speicher geladen):

```
            XmlReader xr = XmlReader.Create("Telefon.xml",myset);
```

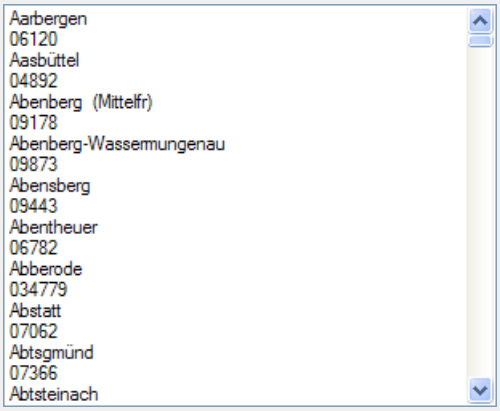
Daten durchlaufen (nur Ebene 3 anzeigen):

```
            while (xr.Read())
                if (xr.Depth == 3) listBox1.Items.Add(xr.Value);
        }
    }
}
```

BEISPIEL 22.52: Einlesen einer XML-Datei mit Telefondaten (Struktur siehe Abbildung) in eine *ListBox*. Es sollen nur die Elemente der dritten Ebene (der Name des Ortes und die Vorwahl) angezeigt werden.

Ergebnis

Das Endergebnis zeigt die folgende Abbildung:



Was und wie wird eigentlich beim *Read* gelesen? Die Antwort ist schnell gefunden, wenn Sie das obige Beispiel etwas abändern:

```
            while (xr.Read())
            {
                Debug.Print(xr.NodeType + " " + xr.Name + " " + xr.Value);
            }
```

Verzichten Sie auf das Filtern und lassen Sie sich zusätzlich *NodeType* und *Name* anzeigen, so erhalten Sie folgende Ausgaben (zur besseren Übersicht haben wir diese etwas eingerückt):

```
XmlDeclaration xml version="1.0" encoding="UTF-8"
Element dataroot
    Element Telefon
        Element Ort
            Text  Aarbergen
        EndElement Ort
        Element Vorwahl
            Text  06120
        EndElement Vorwahl
    EndElement Telefon
    Element Telefon
        Element Ort
            Text  Aasbüttel
        EndElement Ort
        Element Vorwahl
            Text  04892
        EndElement Vorwahl
    EndElement Telefon
...
```

Mit dem Grundwissen aus dem ersten Abschnitt dürften Sie schnell erkennen, dass hier die XML-Struktur auf recht einfache und übersichtliche Weise abgebildet wird. Die Eigenschaft *Depth* ist nur als Hilfestellung vorhanden, damit Sie vor lauter Elementen nicht den Überblick verlieren.

HINWEIS: Seit .NET 2.0 wurden auch diverse Methoden (z.B. *ReadContentAsFloat, ReadContentAsDataTime*) zum direkten Einlesen von Datentypen integriert. Damit bleibt Ihnen das nervtötende nachträgliche Konvertieren erspart.

22.5.5 Erzeugen von XML-Daten mit XmlWriter

Neben den bereits vorgestellten Möglichkeiten, eine XML-Datei zu erzeugen bzw. zu exportieren, bietet sich auch die Verwendung eines *XmlWriters* an. Hierbei handelt es sich um eine recht schnelle und übersichtliche Alternative, die dem Programmierer allerdings eine gewisse Disziplin abverlangt, sind Sie doch selbst dafür verantwortlich, dass es sich um ein wohlgeformtes XML-Dokument handelt. D h., Sie arbeiten auf der Ebene von XML-Elementen, die Sie in der richtigen Reihenfolge und Notation in das neue Dokument einfügen müssen.

BEISPIEL 22.53: Die im vorhergehenden Beispiel in die *ListBox* eingelesenen Werte sollen wieder im XML-Format exportiert werden. Statt Elementen verwenden wir jetzt jedoch Attribute, um eine kompaktere XML-Datei zu erhalten.

C#

```
using System.Xml;
using System.Xml.XPath;
...
```

Objekt erzeugen:

```
    XmlWriter xw = XmlWriter.Create("Export.xml");
```

Prolog erzeugen:

```
    xw.WriteStartDocument();
```

Root-Element schreiben:

```
    xw.WriteStartElement("Telefonnummern");
```

Einzel-Elemente auflisten:

```
    for (int i = 0; i < listBox1.Items.Count; i+=2)
    {
        xw.WriteStartElement("Ort");
```

Werte als Attribute speichern:

```
        xw.WriteAttributeString("Name", listBox1.Items[i].ToString());
        xw.WriteAttributeString("Nummer", listBox1.Items[i+1].ToString());
```

... und nie die schließenden Elemente vergessen!

```
        xw.WriteEndElement();     // für Ort
    }
```

BEISPIEL 22.53: Die im vorhergehenden Beispiel in die *ListBox* eingelesenen Werte sollen wieder im XML-Format exportiert werden. Statt Elementen verwenden wir jetzt jedoch Attribute, um eine kompaktere XML-Datei zu erhalten.

```
    xw.WriteEndElement();         // für Telefonnummern
    xw.WriteEndDocument();
    xw.Close();
```

Vergessen Sie nicht die *Close*-Methode, andernfalls kann es zu Problemen mit der exportierten Datei kommen (Lesefehler).

Ergebnis

Die neue Datei mit geänderter Struktur (*Export.xml*):

```
<?xml version="1.0" encoding="utf-8" ?>
- <Telefonnummern>
    <Ort Name="Aarbergen" Nummer="06120" />
    <Ort Name="Aasbüttel" Nummer="04892" />
    <Ort Name="Abenberg (Mittelfr)" Nummer="09178" />
    <Ort Name="Abenberg-Wassermungenau" Nummer="09873" />
    <Ort Name="Abensberg" Nummer="09443" />
    <Ort Name="Abentheuer" Nummer="06782" />
    <Ort Name="Abberode" Nummer="034779" />
    <Ort Name="Abstatt" Nummer="07062" />
    <Ort Name="Abtsgmünd" Nummer="07366" />
    <Ort Name="Abtsteinach" Nummer="06207" />
    <Ort Name="Abtswind" Nummer="09383" />
    <Ort Name="Abtweiler" Nummer="06753" />
    <Ort Name="Achberg" Nummer="08380" />
```

Wer einen Blick mit *Notepad* in die XML-Datei wagt, wird über das scheinbare Chaos entsetzt sein. Keine Zeilenumbrüche, keine Einrückungen etc. Das alles ist in einer XML-Datei auch nicht unbedingt erforderlich, für eine bessere Lesbarkeit bietet es sich in einigen Fällen dennoch an.

Einfluss auf das Exportformat haben Sie mit Hilfe eines *XmlWriterSettings*-Objekts, das Sie der *Create*-Methode übergeben können.

```
...
          XmlWriterSettings xws = new XmlWriterSettings();
          xws.Indent = true;                  // Einrücken
          xws.IndentChars = "  ";             // zwei Leerzeichen
          XmlWriter xw = XmlWriter.Create("Export.xml", xws);
          xw.WriteStartDocument();
...
```

XML-Ausgabe ohne *XmlWriterSettings*:

```
<?xml version="1.0" encoding="utf-8"?><Telefonnummern><Ort Name="Aarbergen" Nummer="06120" /><Ort
Name="Aasbüttel" Nummer="04892" /><Ort Name="Abenberg  (Mittelfr)" Nummer="09178" /><Ort
Name="Abenberg-Wassermungenau" Nummer="09873" /><Ort Name="Abensberg" Nummer="09443" /><Ort
Name="Abentheuer" Nummer="06782" /><Ort Name="Abberode" Nummer="034779" /><Ort Name="Abstatt"
Nummer="07062" /><Ort Name="Abtsgmünd" Nummer="07366" /><Ort Name="Abtsteinach" Nummer="06207"
/><Ort Name="Abtswind" Nummer="09383" /><Ort Name="Abtweiler" Nummer="06753" /><Ort
Name="Achberg" Nummer="08380" /><Ort Name="Achern" Nummer="07841" /><Ort Name="Achim"
Nummer="04202" /><Ort Name="Achslach" Nummer="09929" /><Ort Name="Acht" Nummer="02656" /><Ort
```

XML-Ausgabe mit *XmlWriterSettings*:

```
<?xml version="1.0" encoding="utf-8"?>
<Telefonnummern>
  <Ort Name="Aarbergen" Nummer="06120" />
  <Ort Name="Aasbüttel" Nummer="04892" />
  <Ort Name="Abenberg  (Mittelfr)" Nummer="09178" />
  <Ort Name="Abenberg-Wassermungenau" Nummer="09873" />
  <Ort Name="Abensberg" Nummer="09443" />
  <Ort Name="Abentheuer" Nummer="06782" />
```

22.5.6 XML transformieren mit XSLT

Nachdem wir Sie mit diversen Möglichkeiten des Schreibens und Lesens von XML-Dokumenten traktiert haben, wollen wir noch kurz auf ein recht interessantes Thema eingehen. Mit Hilfe eines XSLT-Prozessors (*Extensible Style Language Transformation*) und eines entsprechenden XSL-Dokuments können Sie ein XML-Dokument in ein anderes Format transformieren. Dies kann neben einer gefilterten/geänderten XML-Datei auch eine HTML- oder PDF-Datei sein.

BEISPIEL 22.54: Auslesen der Datei *Telefon.xml* und konvertieren in das XML-Format

C#

```
...
using System.Xml;
using System.Xml.Xsl;

namespace WindowsApplication1
{
...
        private void button1_Click(object sender, EventArgs e)
        {
            XslCompiledTransform xslt = new XslCompiledTransform();
            xslt.Load("Telefon.xsl");
            xslt.Transform("Telefon.xml", "Telefon.html");
        }
    }
```

Wer dachte, jetzt jede Menge Code zu produzieren, ist sicher enttäuscht worden. Die Hauptarbeit übernimmt die Transformationsdatei, die alle erforderlichen Steueranweisungen etc. enthält.

BEISPIEL 22.55: Die Transformationsdatei *Telefon.xsl*:

XML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
```

Die wichtigsten HTML-Tags erzeugen:

```
   <HTML>
     <TITLE>Telefonvorwahlen</TITLE>
```

BEISPIEL 22.55: Die Transformationsdatei *Telefon.xsl*:

Tabellenkopf erzeugen:

```
<Table border="1" padding="0" cellspacing="1">
<THEAD>
  <TH>Ort</TH>
  <TH>Vorwahl</TH>
</THEAD>
```

Daten selektieren und in einer Schleife ausgeben:

```
<xsl:for-each select="dataroot/Telefon">
  <tr>
    <td><xsl:value-of select="Ort"/> </td>
    <td><xsl:value-of select="Vorwahl"/></td>
  </tr>
</xsl:for-each>
```

Notwendige HTML-Tags schreiben:

```
    </Table>
   </HTML>
  </xsl:template>
</xsl:stylesheet>
```

Ergebnis

Die erzeugte HTML-Datei im Internet Explorer:

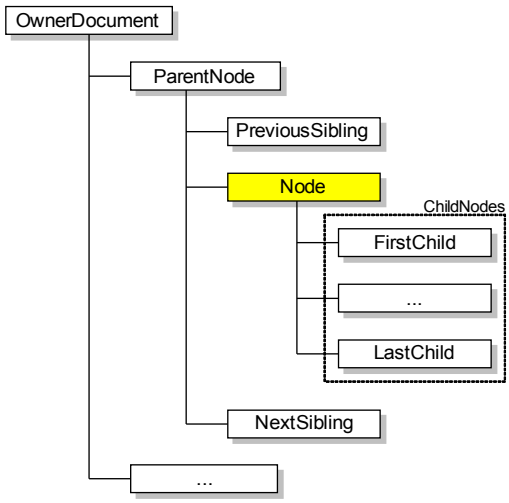
Ort	Vorwahl
Aarbergen	06120
Aasbüttel	04892
Abenberg (Mittelfr)	09178
Abenberg-Wassermungenau	09873
Abensberg	09443
Abentheuer	06782
Abberode	034779
Abstatt	07062
Abtsgmünd	07366
Abtsteinach	06207
Abtswind	09383

22.6 Praxisbeispiele

22.6.1 Mit dem DOM in XML-Dokumenten navigieren

Welche Möglichkeiten, d h. Methoden bzw. Eigenschaften, zur Navigation zwischen den einzelnen Knoten sich bei Verwendung des *XmlDocuments*-Objekts anbieten, zeigt die Skizze auf der folgenden Seite (Ausgangspunkt ist der hervorgehobene Knoten).

Wie Sie sehen, können Sie von jedem beliebigen Knoten aus auf den gesamten Baum zugreifen. Entweder Sie bewegen sich mit *PreviousSibling* bzw. *NextSibling* innerhalb einer Ebene, oder Sie wechseln mit *ParentNode* in die übergeordnete Ebene, um dort ebenfalls mit *PreviousSibling* bzw. *NextSibling* auf die einzelnen Knoten zuzugreifen. Möchten Sie die untergeordneten Elemente eines Knotens verarbeiten, können Sie zunächst mit *FirstChild* auf das erste untergeordnete Element zugreifen, um dann wiederum mit *PreviousSibling* bzw. *NextSibling* mit den weiteren Elementen der dann aktiven Ebene fortzufahren.



Unser kleines Test- und Probierprogramm hat die recht einfache Aufgabe, die Datei *Test7.xml* von der Festplatte zu laden. Nachfolgend soll, ausgehend vom Root-Element, die Navigation zwischen den einzelnen Baumknoten demonstriert werden. Dazu stellen entsprechende Schaltflächen die jeweiligen Methoden zur Verfügung. Ist der Knoten gewechselt, wird die Bezeichnung angezeigt.

Oberfläche

Entwerfen Sie eine einfache Oberfläche nach folgendem Vorbild (Laufzeitansicht):

Quelltext

Zunächst binden Sie den folgenden Namespace ein und definieren einige globale Variablen:

```
using System.Xml;
...
    private XmlDocument xmlDoc = new XmlDocument();
    private XmlElement root;
    private XmlNode node;
```

Mit dem Öffnen des Fensters öffnen Sie auch die Datei und weisen der Variablen *Node* den Root-Knoten zu:

```
    private void Form1_Load(object sender, System.EventArgs e)
    {
      try
      {
        xmlDoc.Load(@"test7.xml");
        root = xmlDoc.DocumentElement;
        node = root;
        label1.Text = node.Name;
      }
      catch
      {  MessageBox.Show("Datei nicht gefunden!"); }
    }
```

Wechseln zum Parent:

```
    private void Button1_Click(object sender, System.EventArgs e)
    {
      if (node.ParentNode != null)
      {
        node = node.ParentNode;
        label1.Text = node.Name;
      }
     else MessageBox.Show("Kein Parent vorhanden!");
    }
```

Wechseln zum Vorgänger in der gleichen Baumebene:

```
private void Button2_Click(object sender, System.EventArgs e)
{
  if (node.PreviousSibling != null)
  {
    node = node.PreviousSibling;
    label1.Text = node.Name;
  }
else  MessageBox.Show("Kein Vorgänger vorhanden!");
}
```

Wechseln zum Nachfolger in der gleichen Baumebene:

```
private void Button3_Click(object sender, System.EventArgs e)
{
  if (node.NextSibling != null)
  {
    node = node.NextSibling;
    label1.Text = node.Name;
  }
  else MessageBox.Show("Kein Nachfolger vorhanden!");
}
```

Wechseln zum ersten Child-Knoten:

```
private void Button4_Click(object sender, System.EventArgs e)
{
  if (node.ChildNodes.Count > 0)
  {
    node = node.FirstChild; label1.Text = node.Name;
  }
  else MessageBox.Show("Keine Untereinträge vorhanden!");
}
```

Wechseln zum letzten Child-Knoten:

```
private void Button5_Click(object sender, System.EventArgs e)
{
  if (node.ChildNodes.Count > 0)
  {
    node = node.LastChild; label1.Text = node.Name;
  }
  else MessageBox.Show("Keine Untereinträge vorhanden!");
}
```

Test

Starten Sie das Programm und versuchen Sie, durch den XML-Baum zu navigieren. Die folgende Abbildung soll Ihnen dabei als Hilfestellung dienen:

HINWEIS: Beachten Sie, dass sich auch über "WELT" noch ein Objekt befindet!

Bemerkungen

- Sicher werden Ihre Versuche mit dem Programm durch reichlich Fehlermeldungen unterbrochen, wenn kein aktives Element bzw. kein gültiges Objekt mehr vorhanden ist. Wir haben absichtlich darauf verzichtet alle eventuellen Fehlerfälle abzufangen, da Sie auf diese Weise viel schneller ein Gefühl für das Navigieren innerhalb des XML-Baums bekommen.
- Mit *!=null* oder mit *try-catch* können Sie zusätzliche Fehlerprüfungen einführen.

22.6.2 XML-Daten in eine TreeView einlesen

Mit einem kleinen Beispielprogramm möchten wir Ihnen zeigen, wie Sie XML-Daten strukturiert in einer *TreeView*-Komponente anzeigen können. Das Programm lässt sich schnell und einfach an Ihre eigenen Erfordernisse anpassen, beispielsweise um den Unterschied zwischen den einzelnen Knotentypen darzustellen.

Oberfläche

Fügen Sie in ein Formular einen *Button* sowie eine *TreeView*-Komponente ein.

Quelltext

Binden Sie zunächst den Namespace *System.Xml* ein.

Mit dem Klick auf die Schaltfläche wird eine Instanz des *XMLDocument*-Objektes erzeugt und die Datei *daten.xml* geladen:

```
private void button1_Click(object sender, System.EventArgs e)
{
    XmlDocument xmlDoc = new XmlDocument();
    try
    {
       xmlDoc.Load("daten.xml");
    }
    catch
    {
       MessageBox.Show("Datei nicht gefunden!");
       return;
    }
```

Gleichzeitig löschen wir die bisherigen Inhalte der *TreeView*-Komponente:

```
    treeView1.Nodes.Clear();
```

Die Prozedur *ShowNode* rufen wir mit folgenden Argumenten auf:

- Vorgängerknoten im Baum (*null*, d.h. keiner) und
- Wurzelknoten der XML-Daten (*xmlDOC*):

```
    ShowNode(null, xmlDoc);
    treeView1.Nodes[0].ExpandAll();
}
```

HINWEIS: Übergeben Sie statt *xmlDOC* das Objekt *xmlDOC.DocumentElement,* so beginnt die Baumansicht direkt mit dem ersten Knoten der XML-Daten, d.h. "WELT", andernfalls mit der Datei selbst.

Die Prozedur zur Anzeige in der *TreeView*:

```
private void ShowNode(TreeNode parent, XmlNode node)
{
    string Caption = "";
    TreeNode tn;
```

Wird kein gültiges DOM-Objekt übergeben, wird die Routine beendet:

```
    if (node == null) return;
```

Die Beschriftung des Baumknotens festlegen:

```
    if (node.NodeType == XmlNodeType.Document)
          Caption = "XML-Datei";
    if (node.NodeType == XmlNodeType.Element)
          Caption = node.Name;
    if ((node.NodeType == XmlNodeType.CDATA)||(node.NodeType == XmlNodeType.Text))
          Caption = node.Value;
    if (Caption == "")
          return;
```

Je nach Knotentyp müssen wir andere Eigenschaften zur Bestimmung der Beschriftung auslesen.

Erzeugen eines neuen Baumknotens mit der gewählten Beschriftung:

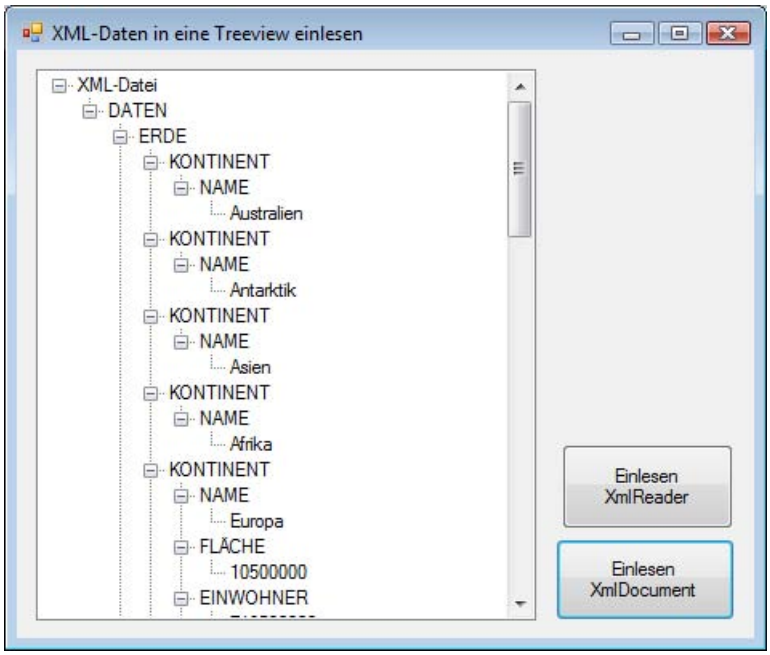
```
    if (parent == null)
        tn = treeView1.Nodes.Add(Caption);
    else
        tn = parent.Nodes.Add(Caption);
```

Sollten Unterelemente vorhanden sein, rufen wir für jedes dieser Elemente die aktuelle Prozedur rekursiv auf:

```
    if (node.ChildNodes != null)
       for (int i = 0; i < node.ChildNodes.Count; i++)
          ShowNode(tn, node.ChildNodes.Item(i));
}
```

Test

Starten Sie das Programm und klicken Sie auf die Schaltfläche, um die Daten aus der Datei zu lesen.



Ergänzung

Wer lieber mit dem *XmlReader* arbeiten möchte, der muss sich zwar von der rekursiven Programmierung verabschieden, der grundsätzliche Ablauf beim Füllen der *TreeView* bleibt jedoch gleich:

Starten der Routine:

```
private void button2_Click(object sender, EventArgs e)
{
    treeView1.Nodes.Clear();
    ShowNode2(null);
    treeView1.Nodes[0].ExpandAll();
}
```

Die eigentliche Routine zum Einlesen:

```
private void ShowNode2(TreeNode parent)
{
    TreeNode tn = parent;
    XmlReaderSettings myset = new XmlReaderSettings();
    myset.IgnoreWhitespace = true;
```

Da unsere XML-Datei eine DTD besitzt, müssen wir deren Verarbeitung vorher zustimmen:

```
    myset.ProhibitDtd = false;
```

Den *XmlReader* initialisieren:

```
    XmlReader xr = XmlReader.Create("daten.xml", myset);
```

Alle XML-Elemente durchlaufen:

```
    while (xr.Read())
    {
```

Handelt es sich um ein *EndElement*, müssen wir im Baum eine Ebene nach "oben" wechseln:

```
        if (xr.NodeType == XmlNodeType.EndElement)
          tn = tn.Parent;
        else
        {
```

Elemente haben einen *Name*:

```
            if (xr.NodeType == XmlNodeType.Element)
            {
                if (tn == null)
                    tn = treeView1.Nodes.Add(xr.Name);
                else
                    tn = tn.Nodes.Add(xr.Name);
            }
```

Texte besitzen einen *Value*:

```
            if (xr.NodeType == XmlNodeType.Text)
                tn.Text = tn.Text + ": " + xr.Value;
        }
    }
}
```

Kapitel 23

Einführung in ADO.NET

Nach wie vor ist ADO.NET die Basis-Datenzugriffstechnologie des .NET-Frameworks. Der Inhalt des vorliegenden Kapitels beschränkt sich auf eine knappe Einführung in die ADO.NET- Technologie, wobei der Schwerpunkt auf dem ersten Teil des Objektmodells, den Datenprovidern, liegt.

Erst das nachfolgende Kapitel 24 widmet sich ausführlich einem der Kernobjekte von ADO.NET, dem *DataSet*. Auf die speziellen Probleme der Datenbindung, d h., auf die Interaktion der Benutzerschnittstelle mit den ADO.NET-Objekten, gehen wir in den speziellen Kapiteln 15, 36 und 42 in Bezug auf den jeweiligen Anwendungstyp näher ein.

HINWEIS: Ausgeklammert werden mussten in diesem Buch die weiterführenden Technologien LINQ to SQL und LINQ to Entities[1], einen kleinen Vorgeschmack bieten das Praxisbeispiel 24.6.5 sowie die Abschnitte 41.7 und 41.8.

23.1 Eine kleine Übersicht

Die umfangreichen Klassenbibliotheken von ADO.NET verlangen vom Einsteiger eine ziemlich steile "Lernkurve".Er tut gut daran, sich zunächst einen Gesamtüberblick zu verschaffen.

23.1.1 Die ADO.NET-Klassenhierarchie

ADO.NET setzt sich aus einer ziemlich komplexen Hierarchie vieler Klassen zusammen. Die daraus erzeugten Objekte lassen sich zunächst in zwei Gruppen aufteilen:

- Datenprovider
- Datenkonsument

Während der *Datenprovider* die Daten zur Verfügung stellt, ist der *Datenkonsument* der Teil der Applikation, welche die Dienste eines Datenproviders nutzt, um auf beliebige Daten zuzugreifen, sie zu lesen, zu speichern und zu ändern.

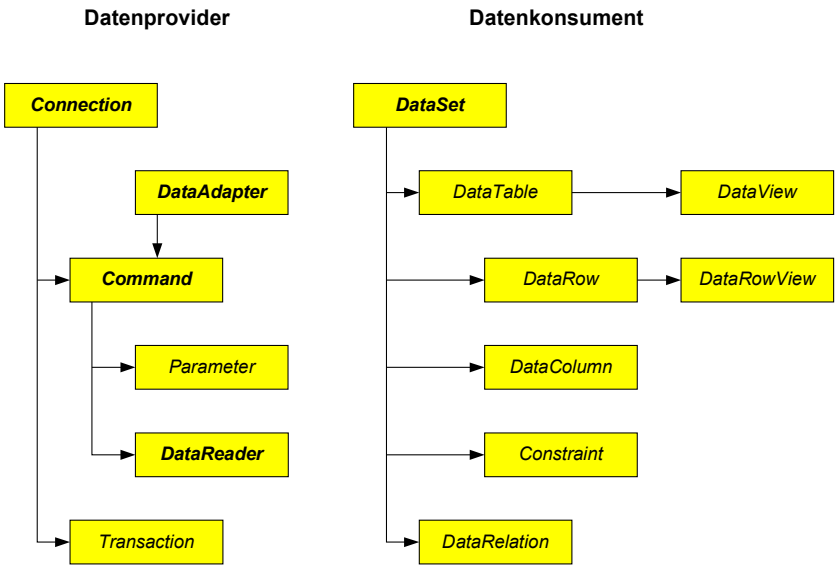
[1] Wir verweisen hier auf unser Buch [Datenbankprogrammierung mit Visual C# 2012].

Die Objekte *Connection*, *Command*, *DataReader* und *DataAdapter* sind die Hauptelemente des .NET-Datenprovider-Modells.

Man bezeichnet die Datenprovider auch als *Verbundene Objekte*, da sie immer in Beziehung zu einer bestimmten Datenquelle stehen. Die Datenkonsumenten hingegen sind *Unverbundene Objekte*, weil sie – ganz im Sinne der ADO.NET-Philosophie – unabhängig von einer Datenquelle ihr völlig autarkes Dasein führen.

Der allen übergeordnete Datenkonsument ist das *DataSet*, es ist gewissermaßen das Kernobjekt von ADO.NET und vergleichbar mit den vom alten ADO her bekannten *Recordset*-Objekten, allerdings ist es weitaus komplexer, da es z.B. mehrere *DataTable*-Objekte und die Beziehungen (Relationen) zwischen ihnen kapseln kann. Ein *DataSet* kann (unter Verwendung eines *DataAdapters*) direkt von der Datenquelle geladen werden, es kann aber auch – ähnlich einem Array – völlig unabhängig von einer Datenbank mit Werten gefüllt werden. Der *DataAdapter* ist quasi das Verbindungsglied zwischen Datenprovider (Datenbank) und Datenkonsument (unverbundene Objekte).

HINWEIS: Um ein erstes praktisches Feeling für die ADO.Net-Klassen zu entwickeln, sollte der Einsteiger bereits jetzt ein einfaches Beispiel ausprobieren, z.B. das PB 23.8.1 "Wichtige ADO.NET-Objekte im Einsatz".



23.1.2 Die Klassen der Datenprovider

Im Einklang mit dem ADO.NET-Objektmodell sind Datenprovider stets in mehrfacher Ausfertigung vorhanden. Die Präfixe charakterisieren die Zugehörigkeit zu einem bestimmten *.NET-Datenprovider,* z.B.:

- *OleDb...*
  Diese Klassen (z.B. *OleDbConnection)* dienen dem OLE Db-Zugriff auf unterschiedlichste Datenbanktypen, für die ein Treiber installiert ist.
- *Sql...*
  Diese Klassen (z.B. *SqlConnection*) dienen dem schnelleren Direktzugriff auf den "hauseigenen" Microsoft SQL Server.

Der *Datenprovider* im .NET Framework kapselt die Datenbank und ermöglicht den Zugriff über eine einheitliche Schnittstelle, er fungiert quasi als Brücke zwischen einer Anwendung und einer Datenbank und wird zum Abrufen von Daten aus einer Datenbank und zum Abgleichen von Änderungen an diesen Daten mit der Datenbank verwendet.

Die Datenquelle selbst kann eine beliebige Struktur haben und sich an einem beliebigen Ort befinden, z.B. eine lokale Access-Datenbank, ein SQL Server oder aber auch verschiedene Adressen des Internets, auf die über Webdienste zugegriffen wird.

.NET-Datenprovider

In der folgenden Tabelle sind wichtige Klassen der *OleDb*- und *SqlServer*-Provider paarweise aufgelistet:

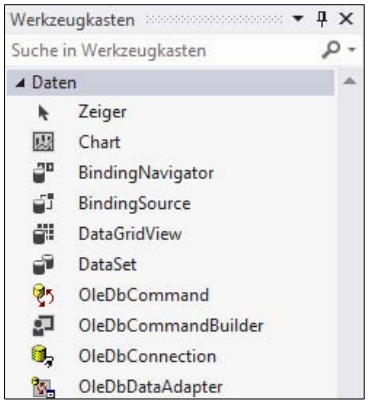
System.Data.OleDb	System.Data.SqlClient	Bedeutung
OleDbConnection	*SqlConnection*	Stellt die Verbindung zur Datenquelle her
OleDbCommand	*SqlCommand*	Führt eine SQL-Abfrage aus
OleDbDataReader	*SqlDataReader*	Ermöglicht einen sequenziellen Nur-Lese-Zugriff auf die Datenquelle
OleDbDataAdapter	*SqlDataAdapter*	Ermöglicht das Füllen eines *DataSet*s mit den Ergebnissen einer SQL-Abfrage
OleDbCommand-Builder	*SqlCommandBuilder*	Erstellt automatisch *Command*-Objekte für die Übernahme der in einem *DataSet* vorgenommenen Änderungen in die Datenbank
OleDbTransaction	*SqlTransaction*	Organisiert die Anwendung von Transaktionen

Weitere Datenprovider

Die Liste der .NET-Datenprovider ist keinesfalls nur auf die nach einer Standardinstallation vorhandenen Provider beschränkt. Neben *OleDb* und *SqlClient* sind in ADO.NET u.a. auch die folgenden Provider enthalten:

- *System.Data.Odbc*
- *System.Data.SqlServerCe*
- *System.Data.OracleClient*

Nach einer Standardinstallation von Visual Studio 2012 werden Sie jedoch noch keine der Datenprovider-Komponenten im Werkzeugkasten vorfinden, was bei der Vielfalt auch nicht sinnvoll wäre. Falls Sie die Komponenten nicht – wie bei den meisten unserer Beispiele – per Code (*new*-Operator), sondern durch Abziehen aus dem Werkzeugkasten erzeugen wollen, wählen Sie das Werkzeugkasten-Kontextmenü *Elemente auswählen...* und fügen Sie die gewünschten Komponenten hinzu:



Anzeige der installierten Datenprovider

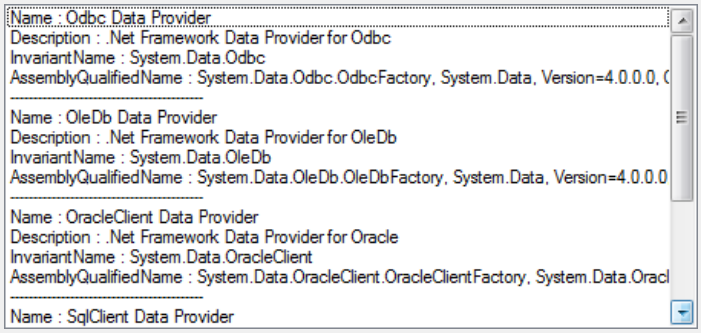
Einen Überblick über alle auf Ihrem System installierten ADO.NET-Datenprovider können Sie mit der Methode *GetFactoryClasses* der *DbProviderFactories*-Auflistung aus dem *System.Data.Common*-Namespace gewinnen.

BEISPIEL 23.1: Alle verfügbaren Datenprovider in einer *ListBox* anzeigen.

C#

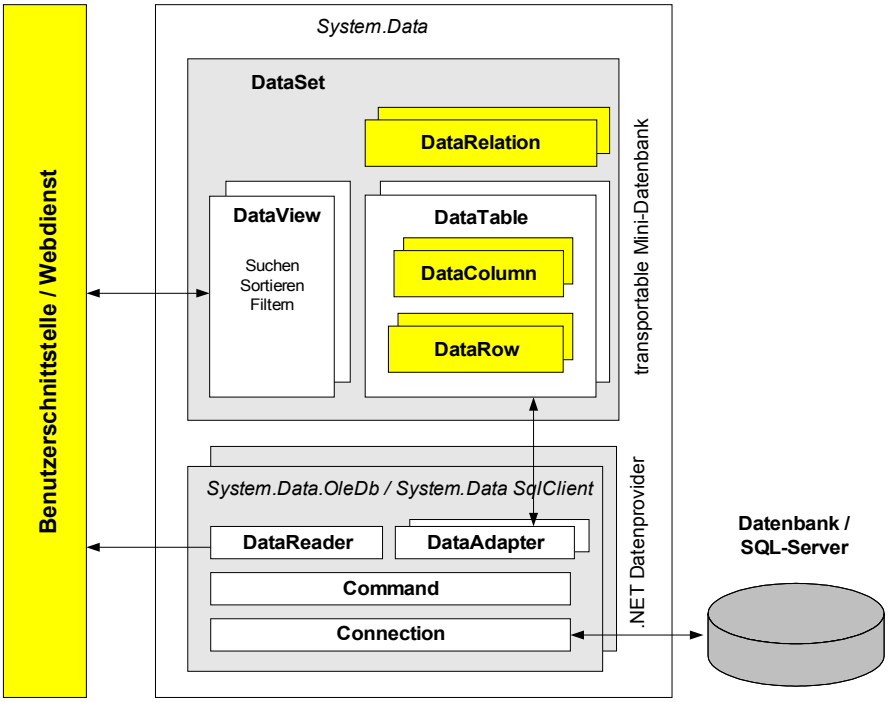
```
DataTable providers = System.Data.Common.DbProviderFactories.GetFactoryClasses();
foreach (DataRow provider in providers.Rows)
{
   foreach (DataColumn col in providers.Columns)
          listBox1.Items.Add(col.ColumnName + " : " + provider[col]);
   listBox1.Items.Add("----------------------------------------");
}
```

Ergebnis



23.1.3 Das Zusammenspiel der ADO.NET-Klassen

In der Abbildung wird versucht, den grundlegenden Zusammenhang zwischen den ADO.NET-Klassen in vereinfachter Form zu verdeutlichen:



Die in obiger Abbildung angegebenen Namespaces (Namensräume) für die ADO.NET- Klassen sind:

- *System.Data*
- *System.Data.OleDb*
- *System.Data.SQLClient*

Das *DataSet* ist vollständig von der Datenbank entkoppelt, denn dazwischen hat sich ein *.NET-Datenprovider* geschoben, der im Bedarfsfall den Datentransport (über die OLE DB- bzw. die direkte SQL Server-Schnittstelle) übernimmt.

HINWEIS: Eine einfache und schnellere Möglichkeit für den Zugriff auf die Datenquelle ist der *DataReader*, da mit ihm auf direktem Wege – also ohne *DataAdapter* und *DataSet* – Daten in die Benutzeroberfläche eingelesen werden können.

23.2 Das Connection-Objekt

Um überhaupt auf eine Datenbank zugreifen zu können, muss als Erstes eine Verbindung zu ihr hergestellt werden. Dazu führt in der Regel kein Weg am *Connection*-Objekt vorbei.

23.2.1 Allgemeiner Aufbau

Der am häufigsten zum Erzeugen und Initialisieren eines *Connection*-Objekts benutzte Konstruktor nimmt einen *ConnectionString* als Parameter entgegen:

```
SYNTAX: Connection conn = new Connection(string ConnectionString);
```

Der *ConnectionString* – die gleichzeitig auch wichtigste Eigenschaft des *Connection*-Objekts – kapselt alle erforderlichen Verbindungsparameter.

Durch Aufruf der (parameterlosen) *Open*-Methode erhält das *Connection*-Objekt eine offene Verbindung aus dem Verbindungspool, falls diese verfügbar ist. Andernfalls wird eine neue Verbindung mit der Datenquelle erstellt.

Nach einer Standardinstallation von Visual Studio 2012 stehen – je nach Auswahl des .NET-Providers – verschiedene *Connection*-Objekte zur Verfügung, z.B.:

- *OleDbConnection*-Objekt
  ... gewährleistet den Zugriff auf eine Vielzahl von Datenquellen, angefangen von einfachen Textdateien über Tabellen bis hin zu kompletten Datenbanken.
- *SqlConnection*-Objekt
  ... ist speziell für die Verwendung mit dem SQL Server optimiert, indem die OLE DB-Schicht umgangen wird.

23.2.2 OleDbConnection

Parameter für OleDb-Zugriff

Die Parameter des *ConnectionString* (bzw. Eigenschaften des *Connection*-Objekts) hängen vom gewählten Datenprovider ab. Die Tabelle zeigt die wichtigsten Angaben für den OLE DB-Zugriff:

Parameter	Bedeutung
Provider	Name des OLE DB-Providers, so wie in der Registrierdatenbank abgelegt (z.B. Microsoft.Jet.OLEDB.4.0 für Microsoft Access)
Data Source	Name der Datenquelle (bei Access-Datenbanken ein Dateiname, z.B. *Nordwind.mdb*)
DSN	Falls auf dem lokalen PC eine Benutzer- oder System-DSN (*Data Source Name*) vorhanden ist, kann auch über diesen Alias auf die Datenbank zugegriffen werden (Angabe von *Data Source* in diesem Fall nicht notwendig)

Parameter	Bedeutung
User	Wenn der Zugriff auf die Datenbank geschützt ist, kann hier der Benutzername angegeben werden
Password	Falls ein User notiert wurde, kann hier das zugehörige Passwort übergeben werden

OleDb-Provider für Access Datenbank

BEISPIEL 23.2: Öffnen einer OLE DB-Verbindung zur Access-Datenbank *Nordwind.mdb*, die sich im aktuellen Anwendungsverzeichnis befindet.

C#
```
using System.Data.OleDb;
...
OleDbConnection conn = new OleDbConnection(
                          "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
conn.Open();
```

Im obigen Beispiel wurde der *ConnectionString* dem *new*-Konstruktor übergeben. Man kann ihn aber auch separat zuweisen, wie das folgende Beispiel zeigt.

BEISPIEL 23.3: Eine zum Vorgängerbeispiel äquivalente Variante

C#
```
using System.Data.OleDb;
...
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;";
conn.Open();
```

Wie Sie erkennen, besteht ein *ConnectionString* aus einer Zeichenfolge mit Attribut/Wert-Paaren für Informationen, die zum Anmelden an eine Datenbank und Zeigen auf eine bestimmte Datenbank erforderlich sind.

HINWEIS: Die Reihenfolge der Parameter im *ConnectionString* ist ohne Bedeutung!

Im obigen Beispiel sind als Minimum nur der (OleDb-)*Provider* und die *DataSource* (Datenquelle) angegeben. Beide Parameter sind wiederum Eigenschaften des *Connection*-Objekts, die man allerdings nur lesen kann (*ReadOnly*).

HINWEIS: Ein *ConnectionString* ist eine Art "Behälter" für die zahlreichen Eigenschaften eines *Connection*-Objekts.

23.2.3 Schließen einer Verbindung

Nachdem die Daten übertragen worden sind, sollte die Verbindung mit Hilfe der *Close*-Methode wieder geschlossen werden. Ansonsten bleibt die Connection weiter geöffnet, auch nachdem die Connection-Instanz selbst terminiert wurde!

Um die Netzbelastung gering zu halten, sollte man – ganz im Sinne der ADO.NET-Philosophie – das Öffnen und Schließen einer Verbindung möglichst innerhalb einer einzigen Befehlskette durchführen.

BEISPIEL 23.4: Es wird kurzzeitig eine Verbindung zur Access-Datenbank *Firma.mdb* geöffnet, um die Kundentabelle in ein DataSet zu übertragen. Danach wird die Verbindung sofort wieder geschlossen.

C#

```
using System.Data.OleDb;
...

OleDbConnection conn = new OleDbConnection(
                           "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Firma.mdb;");
```

Verbindung öffnen:

```
conn.Open();
```

Daten übertragen:

```
DataSet ds = new DataSet();
OleDbDataAdapter da = new OleDbDataAdapter("SELECT * FROM Kunden", conn);
da.Fill(ds, "Kunden");
```

Verbindung schließen:

```
conn.Close();
```

HINWEIS: Obiger Code ist in vielen Fällen nicht ganz unproblematisch, z.B. wenn die Datenbank nicht vorhanden ist. Die saubere Lösung wird im Abschnitt 23.3.5 diskutiert.

23.2.4 Eigenschaften des Connection-Objekts

Da wir im Verlauf dieses Abschnittes bereits viele Eigenschaften des *Connection*-Objekts en passant besprochen haben, soll diese knappe Zusammenfassung den Überblick erleichtern und gleichzeitig einige zusätzliche Informationen liefern.

ConnectionString-Eigenschaft

Diese zweifelsohne wichtigste Eigenschaft des *Connection*-Objekts kapselt sämtliche Verbindungsinformationen zur Datenbank. Außerdem ist es die einzige Eigenschaft, die nicht schreibgeschützt ist (wenn keine Verbindung zur Datenquelle besteht).

Database- und DataSource-Eigenschaft

Was ist der Unterschied zwischen beiden Eigenschaften? Die *DataSource*-Eigenschaft des *Connection*-Objekts entspricht dem *Data Source*-Attribut innerhalb des *ConnectionString*s und enthält den Speicherort der Datenquelle.

Für eine serverbasierte Datenquelle (Microsoft SQL Server, Oracle) bedeutet der Speicherort den Namen des Computers, auf dem der Server installiert ist. Bei dateibasierten Datenbanken, wie z.B. Access, verweist diese Eigenschaft auf den Datenbankpfad (z.B. *c:\Beispiele\Nordwind.mdb*).

Die *Database*-Eigenschaft ist hingegen für Datenquellen, wie z.B. den SQL Server, gedacht, die mehrere Datenbanken unterstützen und entspricht dem Attribut *Initial Catalog* im *ConnectionString*. Beim *SQL Server OleDb-Provider* können wir aber alternativ beide Attributbezeichner verwenden.

BEISPIEL 23.5: Zwei gleichwertige Möglichkeiten.

C#

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Initial Catalog=Northwind;Integrated Security=SSPI";
```

oder

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS; " +
                        "Database=Northwind;Integrated Security=SSPI";
label1.Text = conn.DataSource;              // liefert ".\SQLEXPRESS"
label2.Text = conn.Database;                // liefert "Northwind"
```

Provider-Eigenschaft

Es klingt möglicherweise etwas verwirrend: Während wir unter dem Begriff *.NET-Datenprovider* eine Klassenbibliothek für den Datenzugriff verstehen, ist *Provider* auch eine Eigenschaft des *OleDbConnection*-Objekts.

Die *Provider*-Eigenschaft bezeichnet hier die OLE DB-Schnittstelle, welche die Datenquelle des jeweiligen Herstellers kapselt. Die Tabelle erklärt einige häufig benutzte OLE DB-Schnittstellen:

Datenquelle	Provider-Eigenschaft
Microsoft Access	Microsoft.Jet.OLEDB.4.0
Microsoft SQL Server	SQLOLEDB.1
Microsoft Indexing Service	MSIDXS.1
Oracle	MSDAORA.1

ServerVersion-Eigenschaft

Diese Eigenschaft liefert eine Zeichenfolge zurück, die die Version der Datenbank enthält. Durch Abprüfen von *ServerVersion* können Sie z.B. gewährleisten, dass keine Abfragen an den Server geschickt werden, die von diesem nicht unterstützt werden (z.B. Abfrageergebnisse als XML liefern, was bereits ab SQL Server 2000 möglich ist).

ConnectionTimeout-Eigenschaft

Obwohl diese Eigenschaft schreibgeschützt ist, haben Sie trotzdem die Möglichkeit, innerhalb des *ConnectionString* anzugeben, wie viel Sekunden der OleDb-Provider versuchen soll, die Verbindung zur Datenbank herzustellen.

BEISPIEL 23.6: Zeit bis zum Timeout der Verbindungsaufnahme auf 10 Sekunden festlegen

C#
```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=.\\SQLEXPRESS;...;
                                                    Connect Timeout=10; ... " ;
```

State-Eigenschaft

Diese Eigenschaft liefert den aktuellen Verbindungsstatus. Die möglichen Werte sind Mitglieder der *ConnectionState*-Enumeration.

Konstante	Verbindungszustand
Open	Geöffnet
Closed	Geschlossen
Connecting	Verbindung wird aufgebaut
Executing	Eine Abfrage wird ausgeführt
Fetching	Daten werden abgerufen
Broken	Unterbrochen

23.2.5 Methoden des Connection-Objekts

Open- und Close-Methode

Wenn Sie die *Open*-Methode auf einem bereits geöffneten *Connection*-Objekt ausführen, wird ein Fehler ausgelöst. Hingegen verursacht der Aufruf von *Close* über einer bereits geschlossenen Verbindung keinen Fehler.

HINWEIS: Da Sie standardmäßig mit Verbindungspooling arbeiten, wird die Verbindung nicht wirklich geschlossen, sondern nur zurück an den Pool gesendet.

Es ist keine Vergesslichkeit der Autoren, wenn in manchen Beispielen das *Connection*-Objekt weder mit *Open* geöffnet noch mit *Close* geschlossen wird. Gewissermaßen im Hintergrund können *Fill*- und *Update*-Methode eines *DataAdapter*-Objekts automatisch die Verbindung öffnen (wenn sie nicht schon geöffnet ist) und sie auch wieder schließen, wenn die Operation beendet ist.

ChangeDatabase-Methode

Viele Server, wie z.B. auch der SQL Server, unterstützen mehrere Datenbanken. Mit der *ChangeDatabase*-Methode können Sie die Datenbank zur Laufzeit wechseln, ohne den USE-Befehl verwenden zu müssen.

BEISPIEL 23.7: Zwei äquivalente Varianten zum Wechseln der Datenbank.

C#

```
conn.ChangeDatabase("Northwind");
```

oder

```
OleDbCommand cmd = conn.CreateCommand();
cmd.CommandText = "USE Northwind";
cmd.ExecuteNonQuery();
```

CreateCommand-Methode

Mit dieser Methode können Sie ein neues *Command*-Objekt erzeugen und damit etwas Schreibarbeit einsparen (siehe obiges Beispiel).

23.2.6 Der ConnectionStringBuilder

Um das Zusammenbauen eines *ConnectionString*s etwas übersichtlicher zu gestalten, gibt es die providerspezifische *ConnectionStringBuilder*-Klasse.

BEISPIEL 23.8: Vergleich von zwei Möglichkeiten für das Erstellen einer Verbindungszeichenfolge zur *Northwind*-Datenbank des SQL Servers

C#

```
using System.Data.SqlClient;
```

Ohne *ConnectionStringBuilder*:

```
string connStr = "Data Source = .\\SQLEXPRESS; Initial Catalog=Northwind; Integrated
Security=true";
conn = new SqlConnection(connStr);
```

Mit *ConnectionStringBuilder*:

```
SqlConnectionStringBuilder csb = new SqlConnectionStringBuilder();
csb.DataSource = ".\\SQLEXPRESS";
csb.IntegratedSecurity = true;
csb.InitialCatalog = "Northwind";
```

23.3 Das Command-Objekt

An Abfragen aller Art (SQL-Queries, Stored Procedures) führt beim Programmieren von Datenbankanwendungen kein Weg vorbei. Unter ADO.NET werden für alle Datenbankabfragen *Command*-Objekte benutzt, die zentraler Bestandteil der jeweiligen .NET-Datenprovider sind.

23.3.1 Erzeugen und Anwenden eines Command-Objekts

Wie bei fast allen anderen ADO.NET-Objekten, bieten sich auch zum Erzeugen eines *Command*-Objekts verschiedene Konstruktoren an. Eine übliche Vorgehensweise ist es, die gewünschte Abfrage neben dem zuvor angelegten *Connection*-Objekt an den Konstruktor der Klasse zu übergeben:

SYNTAX:
```
Command cmd = new Command(string sqlCommand, Connection conn);
```

Das so erzeugte und initialisierte *Command*-Objekt kann dann z.B. an den Konstruktor der *DataAdapter*-Klasse weitergereicht werden, um letztendlich ein *DataSet* zu füllen.

Aber es geht auch ohne *DataAdapter* und *DataSet*, denn um SQL-Anweisungen direkt gegen die Datenquelle zu fahren, kann eine der *Execute*-Methoden (*ExecuteNonQuery*, *ExecuteReader*, *ExecuteScalar*) aufgerufen werden.

BEISPIEL 23.9: Erzeugen und Anwenden eines *Command*-Objekts

C#

Es werden zwei *OleDbCommand*-Objekte erstellt. Mit dem ersten werden die Namen der Firmen aller Pariser Kunden aus *Nordwind.mdb* geändert, mit dem zweiten wird ein *DataAdapter* erstellt, der zum Befüllen eines *DataSet*-Objekts mit den Kunden-Datensätzen dient.

```
OleDbConnection conn =
   new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
OleDbCommand updCmd =
   new OleDbCommand("UPDATE Kunden SET Firma = 'Pariser Kunde' WHERE Ort = 'Paris'", conn);
OleDbCommand selCmd =
   new OleDbCommand("SELECT Firma, Kontaktperson, Ort FROM Kunden WHERE Ort = 'Paris'",
                                                                                  conn);

OleDbDataAdapter da = new OleDbDataAdapter(selCmd);
DataSet ds = new DataSet();
conn.Open();
```

UPDATE-Befehl wird gegen die Datenbank gefahren:

```
updCmd.ExecuteNonQuery();
```

DataSet erhält neue Tabelle ("PariserKunden") mit Datensätzen gemäß SELECT-Befehl:

```
da.Fill(ds, "PariserKunden");
conn.Close();
```

HINWEIS: Den vollständigen Code finden Sie im PB 23.8.2 "Eine Aktionsabfrage ausführen".

23.3.2 Erzeugen mittels CreateCommand-Methode

Auch mit Hilfe der *CreateCommand*-Methode eines *Connection*-Objekts können Sie ein *Command*-Objekt erzeugen. Damit ersparen Sie sich etwas Schreibarbeit.

BEISPIEL 23.10: Zwei äquivalente Varianten, wenn ein gültiges *Connection*-Objekt vorliegt.

C#

Variante A:

```
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
```

Variante B:

```
OleDbCommand cmd = conn.CreateCommand();
```

23.3.3 Eigenschaften des Command-Objekts

Wir werden uns auch hier auf eine knappe Darstellung der wichtigsten Eigenschaften beschränken.

Connection- und CommandText-Eigenschaft

Beide Eigenschaften werden üblicherweise bereits im Konstruktor übergeben (siehe oben), man kann sie aber auch separat zuweisen.

BEISPIEL 23.11: Zwei Varianten zum Erzeugen und Initialisieren eines *OleDbCommand*-Objekts

C#

```
OleDbCommand cmd = new OleDbCommand(
                      "UPDATE Kunden SET Firma = 'Pariser Firma' WHERE Ort = 'Paris'",
conn);
```

... ist äquivalent zu:

```
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandText = "UPDATE Kunden SET Firma = 'Pariser Firma' WHERE Ort = 'Paris'";
```

CommandTimeout-Eigenschaft

Um festzulegen, wie lange die Ausführung einer Abfrage maximal dauern darf, können Sie der *CommandTimeout*-Eigenschaft einen Wert in Sekunden zuweisen (Standardwert = 30 Sekunden).

BEISPIEL 23.12: Ein *DataSet* wird mit der Kundenliste der SQL-Datenbank *Northwind* gefüllt, wofür maximal 10 Sekunden zur Verfügung stehen.

C#

```
SqlConnection conn = new SqlConnection(
                   "Data Source=(local);Integrated Security=sspi;Initial Catalog=Northwind");
SqlCommand cmd = new SqlCommand("SELECT CustomerID, CompanyName FROM Customers", conn);Die
Ausführung der Abfrage darf maximal 30 sek dauern:
cmd.CommandTimeout = 30;
SqlDataAdapter da = new SqlDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
conn.Close();
```

HINWEIS: Sollte eine Abfrage dennoch zu lange dauern, so können Sie diese innerhalb einer asynchronen Umgebung mit Hilfe der *Cancel*-Methode abbrechen.

Zum Ausführen einfacher Datenbankabfragen (wie im obigen Beispiel), können Sie auf das explizite Erzeugen eines *Command*-Objekts verzichten, denn man kann den SQL-String auch direkt dem *DataAdapter* als Parameter übergeben.

BEISPIEL 23.13: Eine gleichwertige Realisierung des Vorgängerbeispiels

C#
```
SqlDataAdapter da = new SqlDataAdapter("SELECT CustomerID, CompanyName FROM Customers",
                                                                                conn);
da.SelectCommand.CommandTimeout = 30;
```

CommandType-Eigenschaft

Mit der *CommandType*-Eigenschaft definieren Sie die auszuführende Operation. Mittels der gleichnamigen Enumeration stehen drei Möglichkeiten zur Verfügung:

- *Text* (Standardwert)
  Hier können Sie eine frei definierbare SQL-Abfrage übergeben.
- *StoredProcedure*
  Hier soll eine in der Datenbank gespeicherte Prozedur bzw. Auswahlabfrage aufgerufen werden.
- *TableDirect*
  Hier wird direkt der Name einer Tabelle angegeben (entspricht *SELECT * FROM <Tabellenname>*).

BEISPIEL 23.14: Aufruf der Stored Procedure "Sales by Years" in der Datenbank *Northwind*

C#
```
SqlCommand cmd = new SqlCommand("Sales by Year", conn);
cmd.CommandType = CommandType.StoredProcedure;
SqlParameter parm1 = new SqlParameter("@Beginning_Date", SqlDbType.DateTime);
```

Definition als Input-Parameter:

```
parm1.Direction = ParameterDirection.Input;
```

Das *Beginn*-Datum wird der ersten *TextBox* entnommen:

```
parm1.Value = Convert.ToDateTime(textBox1.Text);
```

Parameter hinzufügen:

```
cmd.Parameters.Add(parm1);
```

HINWEIS: Der nach dem gleichen Prinzip funktionierende Aufruf einer in der Datenbank *Nordwind.mdb* gespeicherten Auswahlabfrage ist im PB 23.8.3 "Eine Auswahlabfrage aufrufen" erklärt.

23.3.4 Methoden des Command-Objekts

ExecuteNonQuery-Methode

Diese Methode setzen Sie vor allem ein, um Aktionsbefehle auf Basis von UPDATE, INSERT oder DELETE direkt gegen die Datenbank auszuführen (also ohne Verwendung von *DataAdapter* und *DataSet*). Rückgabewert ist hier die Anzahl der betroffenen Datensätze (sonst –1).

***BEISPIEL 23.15: ExecuteNonQuery*-Methode**

C# Ein *OleDbCommand*-Objekt wird erzeugt und eine UPDATE-Anweisung gegen die Datenbank gefahren. Die Anzahl betroffener Datensätze wird angezeigt (ein gültiges *OleDbConnection*-Objekt *conn* wird vorausgesetzt).

```
string updStr = "UPDATE Kunden SET Firma = 'Pariser Firma' WHERE Ort = 'Paris'";
OleDbCommand updCmd = new OleDbCommand(updStr, conn);
conn.Open();
```

SQL-Anweisung ausführen und Anzahl betroffener Datensätze anzeigen:

```
label1.Text = cmd.ExecuteNonQuery.ToString();
```

HINWEIS: Das ausführliche Beispiel findet sich im PB 23.8.3 "Eine Aktionsabfrage ausführen".

Weitere Möglichkeiten für Aktionsbefehle sind die Abfrage der Struktur einer Datenbank oder das Erstellen von Datenbankobjekten, wie z.B. Tabellen.

ExecuteReader-Methode

Auf Basis eines SELECT-Befehls erzeugt diese Methode ein *DataReader*-Objekt. Ein Instanziieren des *DataReader*s mittels *new*-Konstruktor entfällt deshalb.

BEISPIEL 23.16: Auf Basis eines *OleDbConnection*-Objekts und eines SELECT-Befehls wird ein *OleDbCommand*-Objekt erstellt und zum Erzeugen eines *DataReader*-Objekts verwendet.

C#
```
string selStr = "SELECT Firma, Kontaktperson, Ort FROM Kunden WHERE Ort = 'Paris'";
OleDbCommand selCmd = new OleDbCommand(selStr, conn);
OleDbDataReader dr;
conn.Open();
dr = selCmd.ExecuteReader(CommandBehavior.CloseConnection);
```

ExecuteScalar-Methode

Rückgabewert dieser Methode ist das Objekt der ersten Spalte der ersten Zeile aus der Menge der zurückgegebenen Datensätze. Generell eignet sich die *ExecuteScalar*-Methode des *Command*-Objekts für alle Abfragen, bei denen man nur an der Rückgabe eines einzigen Wertes interessiert ist.

BEISPIEL 23.17: Abfrage des Namens der Firma eines bestimmten Kunden.

Besonders vorteilhaft kann man *ExecuteScalar* zur Ausführung von Aggregatfunktionen verwenden, was weniger Aufwand erfordert als die Anwendung der *ExecuteReader*-Methode.

BEISPIEL 23.18: Aus der Datenbank *Northwind* wird die Anzahl der in Paris wohnhaften Kunden abgefragt und angezeigt.

C#
```
SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT COUNT(*) AS Anzahl FROM Customers WHERE City = 'Paris'";
cmd.Connection.Open();                                  // oder auch: conn.Open();
label1.Text = cmd.ExecuteScalar().ToString();
cmd.Connection.Close()
```

23.3.5 Freigabe von Connection- und Command-Objekten

In einfachen Codebeispielen stellt man häufig fest, dass der Aufruf von *Dispose()* auf *SqlConnection*- und *SqlCommand*-Objekten fehlt[1]. Auch wurde der Datenzugriffscode nicht in *try-finally*-Blöcke eingerahmt.

BEISPIEL 23.19: Die (leider nicht ganz saubere) Programmierung einer Datenbankverbindung

C#
```
SqlConnection conn = new SqlConnection(connString);
SqlCommand cmd = new SqlCommand(cmdString, conn);
conn.Open();
cmd.ExecuteNonQuery();
conn.Close();
```

Das Problem ist, dass *SqlConnection* und *SqlCommand* die Schnittstelle *IDisposable* implementieren, d h., es können auch Ressourcen aus nicht verwaltetem (unmanaged) Code zu bereinigen sein. Als Programmierer müssen Sie dann unter allen Umständen absichern, dass *Dispose()* auf diesen Objekten aufgerufen wird, nachdem die Arbeit mit ihnen beendet ist. Weil bei Nichtverfügbarkeit der Datenbank immer ein Fehler auftreten kann, sollten Sie den Aufruf von *Dispose()* auch für diesen Fall gewährleisten.

Das Problem lässt sich elegant mittels *using*-Schlüsselwort lösen, welches Ihnen lästige Schreibarbeiten abnimmt, denn intern wird automatisch ein *try-finally*-Block um das entsprechende Objekt generiert und beim Beenden wird für das Objekt *Dispose()* aufgerufen.

BEISPIEL 23.20: Saubere Programmierung des Vorgängerbeispiels

C#
```
using (SqlConnection conn = new SqlConnection(connString))
{
    using (SqlCommand cmd = new SqlCommand(cmdString, conn))
    {
        conn.Open();
        cmd.ExecuteNonQuery();
```

[1] Auch die Codebeispiele dieses Buchs bilden da (aus Platzgründen!) keine Ausnahme.

BEISPIEL 23.20: Saubere Programmierung des Vorgängerbeispiels

```
    }
}
```

Der intern generierte Code für obige Zeilen dürfte etwa folgendermaßen aussehen:

```
SqlConnection conn =  null;
SqlCommand cmd = null;
try
{
    conn = new SqlConnection(connString);
    cmd = new SqlCommand(cmdString, conn);
    conn.Open();
    cmd.ExecuteNonQuery();
}
finally
{
    if (null != cmd) cmd.Dispose();
    if (null != conn) conn.Dispose();
}
```

HINWEIS: Falls Sie, wie im obigen Beispiel, den Aufruf von *Close()* für die *SqlConnection* vermissen, seien Sie trotzdem unbesorgt: Intern überprüft *Dispose()* den Status der Verbindung und schließt diese für Sie.

23.4 Parameter-Objekte

In vielen Fällen enthält ein *Command*-Objekt Parameter bzw. Parameter-Auflistungen, mit denen parametrisierte Abfragen durchführbar sind.

23.4.1 Erzeugen und Anwenden eines Parameter-Objekts

Einer der möglichen Konstruktoren:

SYNTAX:
```
Parameter prm = new Parameter(string pName, DbType pType);
```

Nach dem Zuweisen weiterer Eigenschaften erfolgt das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
cmd.Parameters.Add(Parameter prm );
```

BEISPIEL 23.21: Ein *SqlParameter*-Objekt *p1* wird zur *Parameters*-Collection eines vorhandenen *SqlCommand*-Objekts hinzugefügt.

C#

Im Konstruktor Namen und Datentyp übergeben:

```
SqlParameter p1 = new SqlParameter("@Geburtstag", SqlDbType.DateTime);
```

BEISPIEL 23.21: Ein *SqlParameter*-Objekt *p1* wird zur *Parameters*-Collection eines vorhandenen *SqlCommand*-Objekts hinzugefügt.

Datumswert aus einer *TextBox* zuweisen ...

```
p1.Value = Convert.ToDateTime(textBox1.Text);
```

... und zum *SqlCommand*-Objekt hinzufügen:

```
cmd.Parameters.Add(p1);
```

Die so erzeugten Parameter werden zur Laufzeit in die *CommandText*-Eigenschaft des *Command*-Objekts "eingebaut".

BEISPIEL 23.22: Der definierte Parameter *@Geburtstag* wird in einer SQL-Abfrage eingefügt.

C#
```
cmd.CommandText = "SELECT * FROM Employees WHERE (BirthDate > @Geburtstag)";
```

23.4.2 Eigenschaften des Parameter-Objekts

ParameterName- und Value-Eigenschaft

Beide Eigenschaften dürften selbsterklärend sein.

BEISPIEL 23.23: Eine alternative Zuweisung für das obige erste Beispiel wäre:

C#
```
p1.ParameterName = "@Geburtstag";
p1.Value =  Convert.ToDateTime(textBox1.Text);
```

DbType, OleDbType und SqlDbType-Eigenschaft

Durch das Spezifizieren des Datentyps wird der Wert des Parameters dem Datentyp des .NET-Datenproviders angepasst, bevor er an die Datenquelle weitergereicht wird. Fehlt die Typangabe, so leitet ihn ADO.NET von der *Value*-Eigenschaft des *Parameter*-Objekts ab.

Alternativ zu *OleDbType*- bzw. *SqlDbType*-Eigenschaft kann der Datentyp eines Parameters auch allgemein (generisch) aus *System.Data.DbType* abgeleitet werden.

BEISPIEL 23.24: Ein *Byte*-Parameter wird erzeugt, initialisiert und zur *Parameters*-Collection eines *SqlCommand*-Objekts hinzugefügt.

C#
```
SqlParameter prm = cmd.Parameters.Add("@p2", SqlDbType.TinyInt);
```

Direction-Eigenschaft

Die Eigenschaft bestimmt die Richtung des Parameters relativ zum *DataSet*. Die *ParameterDirection*-Enumeration enthält die in der Tabelle aufgeführten Werte:

ParameterDirection	Beschreibung
Input	Es handelt sich um einen Eingabeparameter (Standard)
InputOutput	Der Parameter unterstützt sowohl Ein- als auch Ausgabe
Output	Es handelt sich um einen Ausgabeparameter
ReturnValue	Der Parameter ist ein Rückgabewert aus einer Operation

BEISPIEL 23.25: Ein *OleDbParameter* wird erstellt und seine *Direction*-Eigenschaft festgelegt.

C#
```
public void CreateOleDbParameter()
{
    OleDbParameter p1 = new OleDbParameter("Description", OleDbType.VarChar, 50);
    p1.IsNullable = true;
    p1.Direction = ParameterDirection.Output;
}
```

23.5 Das CommandBuilder-Objekt

Das manuelle Zuweisen der *Insert-, Update- und DeleteCommand*-Eigenschaften des *DataAdapter*s ist mitunter eine ziemlich aufwändige Angelegenheit. Falls Ihr *DataAdapter* nur auf einer einzigen Datenbanktabelle aufsetzt, können Sie vorteilhaft den *CommandBuilder* zum automatischen Generieren der *Command*-Objekte verwenden.

23.5.1 Erzeugen

Voraussetzung für den Einsatz eines *CommandBuilder*-Objekts ist, dass dem *DataAdapter* vorher die *SelectCommand*-Eigenschaft zugewiesen wurde. Eine einzige Anweisung reicht dann aus, um einen *CommandBuilder* mit einem *DataAdapter* zu verkoppeln:

SYNTAX: `CommandBuilder cmdBuilder = new CommandBuilder(DataAdapter da);`

Der *CommandBuilder* verfolgt nun argwöhnisch alle am *DataSet* vorgenommenen Änderungen und generiert die erforderlichen Queries bzw. *Command*-Objekte selbstständig im Hintergrund.

23.5.2 Anwenden

Die *Update*-Methode des *DataAdapter*s würde im folgenden Beispiel ohne *OleDbCommandBuilder* fehlschlagen.

BEISPIEL 23.26: Aktualisieren der Kunden-Tabelle aus *Nordwind.mdb*.

C#

Beim Instanziieren erhält der *DataAdapter* automatisch auch seine *SelectCommand*-Eigenschaft, sodass diese nicht explizit zugewiesen werden muss:

```
OleDbDataAdapter da = new OleDbDataAdapter("SELECT * FROM Kunden", conn);
```

BEISPIEL 23.26: Aktualisieren der Kunden-Tabelle aus *Nordwind.mdb*.

Ein *OleDbCommandBuilder* wird mit dem *OleDbDataAdapter* verbunden:

```
OleDbCommandBuilder cmdB = new OleDbCommandBuilder(da);
```

Ein *DataSet* wird mit den Daten gefüllt:

```
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Kunden");
...
```

Nachdem die Daten geändert wurden, werden die Änderungen in die Datenbank zurückgeschrieben:

```
da.Update(ds, "Kunden");
```

Beim Aufruf von *Dispose* wird die Zuordnung von *CommandBuilder* zu *DataAdapter* aufgehoben, und die generierten Befehle werden nicht mehr verwendet.

HINWEIS: Ein Beispiel für den Einsatz des *CommandBuilder*s finden Sie im PB 23.8.4 "Die Datenbank aktualisieren".

23.6 Das DataReader-Objekt

Häufig genügt ein Lesezugriff auf die Datensätze. Dabei müssen im Frontend meist nur einige für die Listendarstellung benötigte Komponenten gefüllt bzw. aktualisiert werden (*ListBox*, *ComboBox*, *ListView*, *TreeView*, *DataGridView* usw.).

Im .NET-Framework gibt es für diesen Zweck den *DataReader*. Diese Klasse ist für einen einmaligen ReadOnly-Hochgeschwindigkeitszugriff auf eine Datensatzgruppe optimiert und ähnelt anderen *Reader*-Objekten wie *TextReader*, *StreamReader* und *XmlReader*. In Abhängigkeit vom verwendeten .NET-Datenprovider gibt es auch hier unterschiedliche Typen (*SqlDataReader*, *OleDbDataReader*).

23.6.1 DataReader erzeugen

Einen *DataReader* erzeugt man in der Regel nicht mit dem *new*-Konstruktor, sondern mit der *ExecuteReader*-Methode des zugrunde liegenden *Command*-Objekts:

SYNTAX: `DataReader dr = cmd.ExecuteReader();`

Mitunter wird auch dem *Execute*-Konstruktor als Argument der Wert *CloseConnection* (aus der *CommandBehavior*-Enumeration) übergeben. Damit ist gewährleistet, dass die Verbindung automatisch nach dem Durchlauf des *DataReader*s geschlossen wird.

BEISPIEL 23.27: Ein *DataReader* wird instanziiert.

C#
```
DataReader dr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

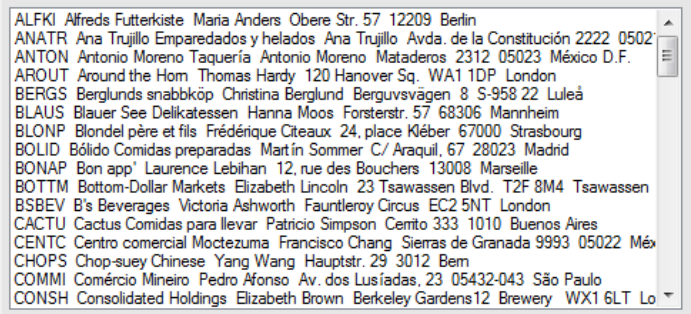
23.6.2 Daten lesen

Das Auslesen der Informationen innerhalb einer Schleife ist typisch für die Arbeit mit dem *DataReader*.

BEISPIEL 23.28: Die Kundentabelle aus *Nordwind.mdb* wird zeilenweise ausgelesen.

C#
```
const string SQL = "SELECT * FROM Kunden ORDER BY KundenCode";
OleDbConnection conn = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
                                                    Data Source=Nordwind.mdb;");
OleDbCommand cmd = new OleDbCommand(SQL, conn);
OleDbDataReader dr;
conn.Open();
dr = cmd.ExecuteReader();
string str;
string tab = "  ";
while (dr.Read())
{
  str = dr["KundenCode"] + tab;
  str += dr["Firma"] + tab;
  str += dr["Kontaktperson"] + tab;
  str += dr["Strasse"] + tab;
  str += dr["PLZ"] + tab;
  str += dr["Ort"];
  listBox1.Items.Add(str);
}
dr.Close();
conn.Close();
```

Ergebnis



HINWEIS: Es ist wichtig, dass Sie den *DataReader* so schnell wie möglich nach dem Auslesen der Daten wieder schließen, da sonst das *Connection*-Objekt blockiert ist!

23.6.3 Eigenschaften des DataReaders

Item-Eigenschaft

Diese Eigenschaft ermöglicht den Zugriff auf die aktuelle Spalte, der Rückgabewert ist vom *Object*-Datentyp (ähnlich der *Item*-Eigenschaft des *DataRow*-Objekts). Falls der Datentyp vorher bekannt ist, sollte man eine der *Get*-Methoden (siehe unten) für den Zugriff verwenden.

FieldCount-Eigenschaft

Diese Eigenschaft liefert die Gesamtanzahl der Datensätze.

IsClosed-Eigenschaft

Der Wert ist *true*, falls der *DataReader* geschlossen ist.

23.6.4 Methoden des DataReaders

Read-Methode

Damit wird das automatische Weiterbewegen zum nächsten Datensatz innerhalb der *while*-Schleife ermöglicht (Rückgabewert *true/false*).

GetValue- und GetValues-Methode

Während *GetValue* – ähnlich der *Item*-Eigenschaft – den Wert einer Spalte (basierend auf dem Spaltenindex) zurückgibt, nimmt *GetValues* ein Array entgegen, in welchem der *DataReader* den Inhalt der aktuellen Zeile ablegt. Mit *GetValues* wird beste Performance erreicht.

GetOrdinal- und ähnliche Methoden

Eine Vielzahl von *Get*...-Methoden ermöglichen ein Konvertieren der gelesenen Werte in fast jeden Datentyp.

BEISPIEL 23.29: Ein Datumswert aus der *Employee*-Tabelle wird ausgelesen.

C#
```
DateTime aDate;
aDate = dr.GetDateTime(dr.GetOrdinal("BirthDate"));
```

Bemerkungen

Nach dem Laden der Daten mit dem *DataReader* kopiert man in der Regel die Datensätze zeilenweise in Objekte um.

BEISPIEL 23.30: Einlesen von Kundendaten

C#
```
CKunde kd = new CKunde();
kd.ID = Convert.ToInt64(DataReader["id"]);
```

BEISPIEL 23.30: Einlesen von Kundendaten

```
k.Ort = Convert.ToString(Ort["id"]);
```

Das ist insbesondere bei vielen Tabellenspalten sehr arbeitsaufwändig. Außerdem liegt die meiste Arbeit noch vor Ihnen, denn in der Regel wollen Sie die Daten nicht nur lesen, sondern Sie wollen auch Änderungen speichern.

23.7 Das DataAdapter-Objekt

Datenadapter werden in einer Art "Brückenfunktion" dazu genutzt, Daten mittels SQL-Anweisungen aus Datenquellen in *DataSet*s zu transportieren bzw. um Datenquellen mit den geänderten Inhalten von *DataSet*s zu aktualisieren. Das *DataAdapter*-Objekt verwendet das *Connection*-Objekt des jeweiligen .NET-Datenproviders, um eine Verbindung zu einer Datenquelle herzustellen, und ist außerdem auf verschiedene *Command*-Objekte angewiesen.

Hin- und Rücktransport der Daten zwischen Datenquelle und *DataSet* werden mit der *Fill*- und *Update*-Methode des *DataAdapter*s realisiert. Beide lösen die entsprechenden SQL-Anweisungen aufgrund der dem *DataAdapter* übergebenen *Command*-Objekte aus.

23.7.1 DataAdapter erzeugen

Mehrere überladene Konstruktoren stellen den Newcomer vor die Qual der Wahl.

Konstruktor mit SELECT-String und Connection-Objekt

Im einfachsten Fall kommt man sogar ohne *Command*-Objekt aus, es genügt, dem Konstruktor des *DataAdapter*-Objekts eine SELECT-Anweisung und das *Connection*-Objekt als Parameter zu übergeben:

SYNTAX: `DataAdapter da = new DataAdapter(string selectStr, Connection conn);`

BEISPIEL 23.31: Ein *DataAdapter* füllt ein *DataSet* mit Datensätzen aus *Nordwind.mdb*.

C#
```
using System.Data.OleDb;
...
OleDbConnection conn = new OleDbConnection(
                           "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
OleDbDataAdapter da = new OleDbDataAdapter("SELECT * FROM Kunden WHERE Ort = 'Paris'", conn);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Konstruktor mit SelectCommand-Objekt

Eine weitere Möglichkeit ist die Verwendung eines Konstruktors, dem ein *Command*-Objekt (SELECT-Befehl) zu übergeben ist:

SYNTAX: `DataAdapter da = new DataAdapter(Command selectCommand);`

BEISPIEL 23.32: Das Vorgängerbeispiel wird mit einem *Command*-Objekt realisiert.

C#
```
using System.Data.OleDb;
...
OleDbConnection conn = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data
Source=Nordwind.mdb;");
OleDbCommand cmd = new OleDbCommand("SELECT Firma FROM Kunden WHERE Ort = 'Paris'");
cmd.Connection = conn;
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

23.7.2 Command-Eigenschaften

Ein *DataAdapter* benötigt für die komplette Zusammenarbeit mit der Datenquelle vier verschiedene *Command*-Objekte, die als Eigenschaften zugewiesen werden:

- *SelectCommand* zur Abfrage
- *UpdateCommand* zur Aktualisierung
- *InsertCommand* zum Einfügen
- *DeleteCommand* zum Löschen

BEISPIEL 23.33: Realisierung der Vorgängerbeispiele mittels *SelectCommand*-Eigenschaft

C#
```
using System.Data.OleDb;
...
OleDbConnection conn = new OleDbConnection( "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=Nordwind.mdb;")
OleDbCommand cmd = new OleDbCommand("SELECT Firma FROM Kunden WHERE Ort = 'Paris'");
cmd.Connection = conn;
OleDbDataAdapter da = new OleDbDataAdapter();
da.SelectCommand = cmd;
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "PariserKunden");
conn.Close();
```

Die *SelectCommand*-Eigenschaft muss gesetzt werden, **bevor** die *Fill*-Methode des *DataAdapter*s aufgerufen wird.

23.7.3 Fill-Methode

Die relativ unkomplizierte *Fill*-Methode des *DataAdapter* hatten Sie bereits in zahlreichen Beispielen kennen gelernt. Hier noch einmal die am häufigsten benutzte Aufrufvariante:

SYNTAX: `DataAdapter da.Fill(DataSet ds, string tblName)`

BEISPIEL 23.34: Ein *DataSet* wird mit der Kundentabelle aus *Nordwind.mdb* gefüllt. Im *DataSet* sollen die Namen aller Firmen aus Paris geändert werden in "Pariser Firma".

C#
```
OleDbConnection conn = new OleDbConnection(
                            "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
OleDbDataAdapter da = new OleDbDataAdapter("SELECT * FROM Kunden", conn);
DataSet ds = new DataSet();
da.Fill(ds, "Kunden");
```

Das Arbeiten mit den Daten im *DataSet*:

```
DataTable dt = ds.Tables["Kunden"];
foreach (DataRow cRow in dt.Rows)            // alle Zeilen der DataTable durchlaufen
    if (cRow["Ort"].ToString() == "Paris")
       cRow["Firma"] = "Pariser Firma";
...
```

Das Beispiel wird im folgenden Abschnitt fortgesetzt!

Begrenzen der Datenmenge

Geht es nur um die Übertragung kleinerer Datenmengen, so ist die bislang praktizierte Vorgehensweise problemlos, nicht aber wenn es sich um Hunderte von Datensätzen handelt.

Abhilfe schafft eine (überladene) Version der *Fill*-Methode, die die Anzahl der zu transportierenden Datensätze begrenzt:

SYNTAX: `int z = DataAdapter da.Fill(DataSet ds, int start, int anzahl, string tblName);`

start = Nummer der Startzeile
anzahl = Anzahl der abzurufenden Datensätze
z = Anzahl der tatsächlich zurückgegebenen Datensätze

BEISPIEL 23.35: Ab Zeile 100 werden 50 Zeilen aus der Datenbank abgerufen und in die "Kunden"-Tabelle gefüllt.

C#
```
int z = da.Fill(ds, 100, 50, "Kunden");
```

23.7.4 Update-Methode

Irgendwann einmal müssen die im *DataSet* vorgenommenen Änderungen in die Datenquelle zurückgeschrieben werden. Zu diesem Zweck wird (kurzzeitig) eine Verbindung zur Datenbank aufgebaut. Genauso wie beim Füllen spielt auch hier ein *DataAdapter*-Objekt die Vermittlerrolle, wobei dessen *Update*-Methode gewissermaßen das Pendant zur *Fill*-Methode ist und zum Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenquelle dient.

Genauso wie die *Fill*-Methode benötigt die *Update*-Methode als Parameter die Instanz eines *DataSet*s und (optional) den Namen der *DataTable*.

SYNTAX: `DataAdapter da.Update( DataSet ds, string tblName);`

Bei der *Update*-Methode läuft es nicht ganz so einfach ab wie bei der *Fill*-Methode, da ein *DataSet* völlig autark existiert und nur gelegentlich mit der Datenbank verbunden wird.

BEISPIEL 23.36: (*Fortsetzung*)

C#

Ziel ist das Zurückschreiben der in der Spalte "Firma" (und nur dort!) vorgenommenen Änderungen in die Datenquelle. Grundlage ist eine UPDATE-Anweisung mit zwei Parametern (die? sind die Platzhalter):

```
...
OleDbCommand cmd = new OleDbCommand("UPDATE Kunden SET Firma = ? WHERE KundenCode = ?",
conn);
```

Der *Add*-Methode werden Parametername, Datentyp, Spaltenbreite und Spaltenname übergeben:

```
cmd.Parameters.Add("@p1", OleDbType.VarChar, 30, "Firma");
```

Für die Schlüsselspalte eine Extrawurst:

```
OleDbParameter prm = cmd.Parameters.Add("@p2", OleDbType.VarChar);
prm.SourceColumn = "KundenCode";
```

Der ursprüngliche Wert (beim Füllen des *DataSet*s) ist maßgebend:

```
prm.SourceVersion = DataRowVersion.Original;

da.UpdateCommand = cmd;
da.Update(ds, "Kunden");
```

HINWEIS: Der Kern der Aktualisierungslogik liegt in der WHERE-Bedingung der UPDATE-Anweisung. Der Datensatz wird nur dann aktualisiert, wenn der Wert der Schlüsselspalte, mit dem er geladen wurde, noch vorhanden ist.

- Durch Einsatz eines *CommandBuilder*-Objekts kann das manuelle Erstellen der *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Eigenschaften automatisiert werden.
- Ein komplettes Beispiel finden Sie im PB 23.8.4 "Die Datenbank aktualisieren"

23.8 Praxisbeispiele

23.8.1 Wichtige ADO.NET-Objekte im Einsatz

Wer sich nicht nur blind auf die Hilfe von Assistenten verlassen möchte, sollte sich in der ADO.NET-Objekthierarchie ein wenig auskennen, damit er die Objekte bei Bedarf selbst per Code programmieren kann.

Die *Columns*- und *Rows*-Auflistungen zählen zu den wichtigsten Eigenschaften der *DataTable*-Klasse, weil sie den Zugriff auf sämtliche Spalten und Zeilen der Tabelle ermöglichen. Das vorliegende Beispiel soll das Zugriffsprinzip verdeutlichen, indem es uns den Inhalt der *Artikel*-Tabelle der *Nordwind*-Datenbank anzeigt.

Oberfläche

Sie brauchen lediglich eine *ListBox* und einen *Button* zum Beenden (siehe Laufzeitansicht).

Quellcode

```
using System.Data.OleDb;
...
```

Alles beginnt mit der Festlegung der Verbindungszeichenfolge (*ConnectionString*) zur Access-Datenbank, die wir aus Bequemlichkeitsgründen gleich mit in das Anwendungsverzeichnis kopiert haben, um nicht den kompletten Datenbankpfad eintragen zu müssen:

```
    private void Form1_Load(object sender, System.EventArgs e)
    {
        OleDbConnection conn =
            new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb");
        string cmdStr =
           "SELECT ArtikelNr,Artikelname,Liefereinheit,Einzelpreis,Mindestbestand FROM Artikel";
        OleDbCommand cmd = new OleDbCommand(cmdStr, conn);
```

Nun geht es um das Füllen des *DataSets* mit Hilfe des *DataAdapter*:

```
        OleDbDataAdapter da = new OleDbDataAdapter(cmd);
        DataSet ds = new DataSet();
        conn.Open();
        da.Fill(ds, "ArtikelListe");
        conn.Close();
```

Die Datenbankverbindung ist ab jetzt wieder getrennt und der Benutzer arbeitet mit dem abgekoppelten *DataSet* quasi wie mit einer Minidatenbank:

```
        DataTable dt = ds.Tables["ArtikelListe"];
```

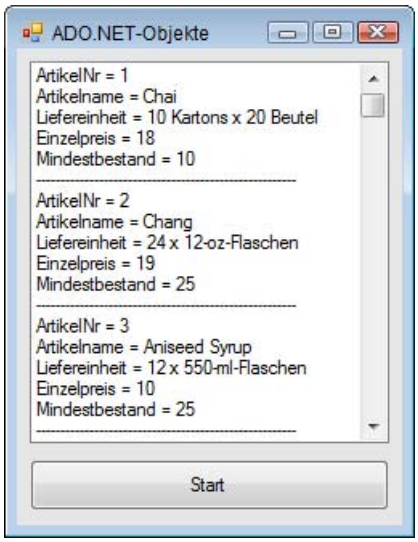
Nachdem je eine Zeilen- und Spaltenvariable definiert sind, sorgen zwei ineinander verschachtelte *foreach*-Schleifen für den Durchlauf der Auflistungen:

```
        foreach (DataRow cRow in dt.Rows)
```

```
        {
            foreach (DataColumn cCol in dt.Columns)
                listBox1.Items.Add(cCol.ColumnName + " = " + cRow[cCol.Ordinal].ToString());
            listBox1.Items.Add("--------------------------------------------------");
        }
    }
```

Test

Sofort nach Programmstart erscheint der Inhalt der *Artikel*-Tabelle in der *ListBox*.



Bemerkungen

- Am Quellcode können Sie den typischen Ablauf eines Datenbankzugriffs studieren: Verbindung öffnen, Daten übertragen, Verbindung schließen.
- Beim Durchlaufen der Datensätze werden Sie die vom altvertrauten ADO-*Recordset*-Objekt her bekannten Methoden wie *MoveFirst*, *MoveNext* etc. vergeblich suchen. Dafür besteht unter ADO.NET keinerlei Notwendigkeit mehr, da alle Datensätze im *DataSet* quasi wie in einem Array gespeichert sind und ein sofortiger (indizierter) Zugriff möglich ist, ohne dass man sich erst mühsam "hinbewegen" muss.

23.8.2 Eine Aktionsabfrage ausführen

Wir wollen an die *Nordwind.mdb*-Beispieldatenbank folgendes SQL-Statement absetzen:

```
UPDATE Kunden SET Firma = 'Londoner Firma' WHERE Ort = 'London'
```

Das vorliegende Beispiel zeigt, wie Sie dazu die *ExecuteNonQuery*-Methode des *Command*-Objekts verwenden können.

Oberfläche

Sie brauchen ein *DataGridView*, zwei *TextBox*en, zwei *Buttons* und einige *Labels* (siehe Laufzeitansicht). Beide *TextBox*en sollen dazu dienen, dass Sie die Einträge für den Firmennamen und den Ort zur Laufzeit verändern können.

Quellcode

Für das Ausführen des Beispiels wären eigentlich ein *Connection*- und ein *Command*-Objekt völlig ausreichend. Da wir uns aber auch von der Wirkung des UPDATE-Befehls überzeugen wollen, müssen wir einigen zusätzlichen Aufwand für die Anzeige betreiben: Das *DataGridView* benötigt ein *DataSet* als Datenquelle, welches wiederum von einem *DataAdapter* gefüllt wird.

```
using System.Data.OleDb;
public partial class Form1 : Form
{
    OleDbConnection conn =
                new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
                                                                Data Source=Nordwind.mdb;");
    DataSet ds = new DataSet();
    OleDbCommand cmd = new OleDbCommand();
```

Aktionsabfrage starten:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        OleDbDataAdapter da =
           new OleDbDataAdapter("SELECT Firma, Kontaktperson, Ort FROM Kunden ORDER BY Firma",
                                conn);
        ds.Clear();
        cmd.Connection = conn;
```

Das Zusammenbasteln des UPDATE-Strings verlangt etwas Fingerspitzengefühl, darf man doch auch die Apostrophe ('), die die Feldbezeichner einschließen, nicht vergessen:

```
        cmd.CommandText = "UPDATE Kunden SET Firma = '" + textBox1.Text + "' WHERE Ort = '" +
                          textBox2.Text + "'";
```

Sicherheitshalber haben wir diesmal den kritischen Programmteil in eine Fehlerbehandlungsroutine eingebaut:

```
        try
        {
             conn.Open();
```

Die folgende Anweisung führt den UPDATE-Befehl aus und zeigt gleichzeitig die Anzahl der in der Datenbank geänderten Datensätze an:

```
             label1.Text = cmd.ExecuteNonQuery().ToString();
        }
        catch (Exception ex)
        {
```

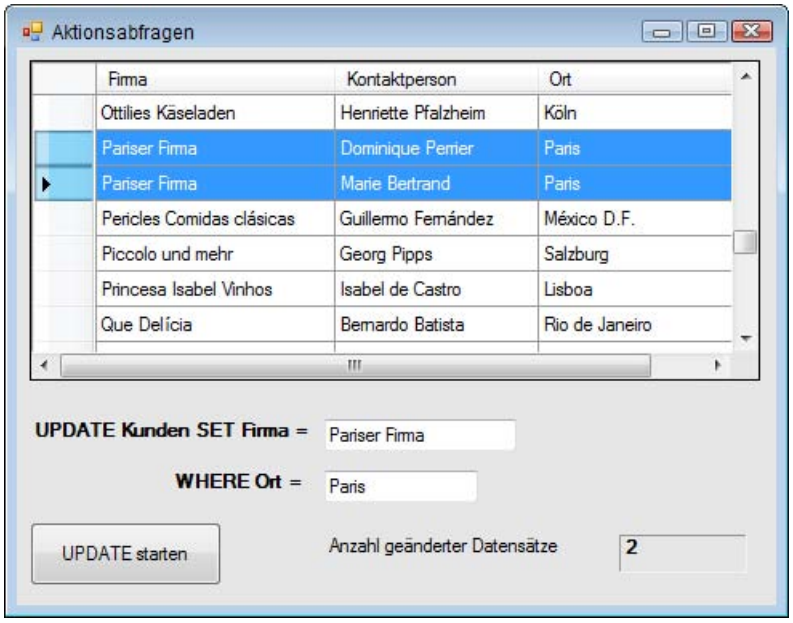
```
            MessageBox.Show(ex.Message);
        }
        da.Fill(ds, "Kunden");
        conn.Close();
```

Das *DataGridView* an das *DataSet* anklemmen:

```
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "Kunden";
    }
}
```

Test

Stimmt die Verbindungszeichenfolge des *Connection*-Objekts, dürfte es keine Probleme beim Ausprobieren unterschiedlicher Updates geben.



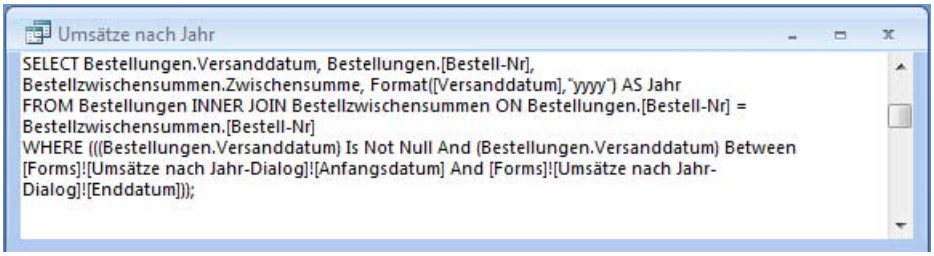
Bemerkungen

- Bei SQL-Aktionsabfragen werden keine Datensätze gelesen bzw. zurückgeliefert, sondern es geht lediglich um direkte Änderungen in der Datenquelle per SQL-Befehl (UPDATE, INSERT, DELETE). Ein *DataSet* ist dabei nicht beteiligt!
- Wie Sie die Änderungen zuerst in einer *DataTable* vornehmen und erst danach in die Datenbank zurückschreiben, erfahren Sie im PB 23.8.4 "Die Datenbank aktualisieren".

23.8.3 Eine Auswahlabfrage aufrufen

Die unter Microsoft Access gespeicherten Auswahlabfragen kann man quasi als Pendant zu den Stored Procedures des Microsoft SQL Servers betrachten. Öffnen Sie das Datenbankfenster von *Nordwind.mdb* und Sie sehen das zahlreiche Angebot an vorbereiteten Abfragen, die Sie natürlich auch selbst um weitere ergänzen können:

Hinter jeder Auswahlabfrage verbirgt sich in der Regel eine parametrisierte SQL-SELECT-Anweisung, die Sie sich im Access-Datenbankprogramm durch Öffnen der Entwurfsansicht über den Kontextmenübefehl *SQL-Ansicht* anschauen können. Dabei finden Sie auch die zu übergebenden Parameter und deren Datentypen leicht heraus:



Oberfläche

Ein *DataGridView*, zwei *TextBox*en und ein *Button* sollen für unseren Test genügen (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.OleDb;
public partial class Form1 : Form
{
```

```
...
   private void button1_Click(object sender, System.EventArgs e)
   {
       string connStr = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;";
       OleDbConnection conn = new OleDbConnection(connStr);

       OleDbCommand cmd = new OleDbCommand("[Umsätze nach Jahr]", conn);
       cmd.CommandType = CommandType.StoredProcedure;
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
      OleDbParameter parm1 = new OleDbParameter("@Anfangsdatum", OleDbType.DBDate);
      parm1.Direction = ParameterDirection.Input;
      parm1.Value = Convert.ToDateTime(textBox1.Text);
      cmd.Parameters.Add(parm1);

      OleDbParameter parm2 = new OleDbParameter("@EndDatum", OleDbType.DBDate);
      parm2.Direction = ParameterDirection.Input;
      parm2.Value = Convert.ToDateTime(textBox2.Text);
      cmd.Parameters.Add(parm2);
```

Das *Command*-Objekt wird dem Konstruktor des *DataAdapter*s übergeben. Nach dem Öffnen der *Connection* wird die Abfrage ausgeführt. Die zurückgegebenen Datensätze werden in einer im *DataSet* neu angelegten Tabelle mit einem von uns frei bestimmten Namen *Jahresumsätze* gespeichert:

```
      OleDbDataAdapter da = new OleDbDataAdapter(cmd);
      DataSet ds = new DataSet();
      try
      {
          conn.Open();
          da.Fill(ds, "Jahresumsätze");
          conn.Close();
      }
      catch(Exception ex)
      {
          MessageBox.Show(ex.ToString());
      }
```

Die Anzeige:

```
      dataGridView1.DataSource = ds;
      dataGridView1.DataMember = "Jahresumsätze";
```

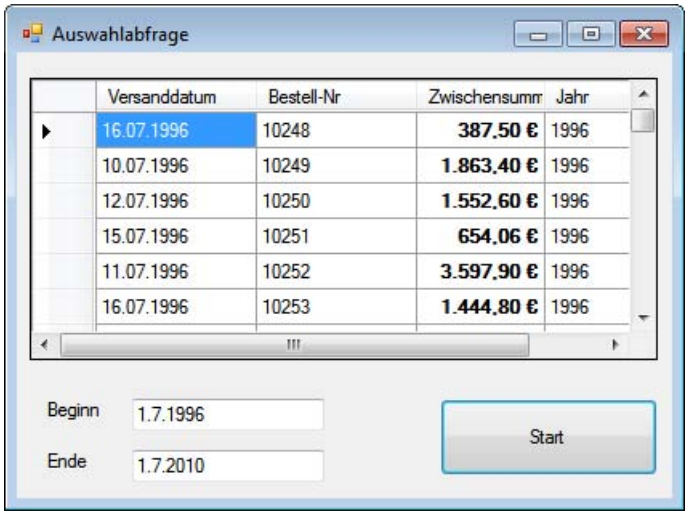
Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
     dataGridView1.Columns.Remove("Zwischensumme");
     DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
     tbc.DataPropertyName = "Zwischensumme";
     tbc.HeaderText = "Zwischensumme";
```

```
      tbc.Width = 80;
      tbc.DefaultCellStyle.Format = "c";
      tbc.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
      tbc.DefaultCellStyle.Font = new Font(dataGridView1.Font, FontStyle.Bold);
      tbc.DisplayIndex = 2;
      dataGridView1.Columns.Add(tbc);
    }
}
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



23.8.4 Die Datenbank aktualisieren

Aktualisieren (UPDATE), Hinzufügen (INSERT) und Löschen (DELETE) von Datensätzen zählen zu den kritischen Datenbankoperationen, die auch unter ADO.NET weitaus mehr Aufmerksamkeit erfordern, als eine einfache SELECT-Abfrage.

Ganz im Einklang mit der ADO.NET-Philosophie müssen wir dabei in drei Etappen vorgehen:

- Das *DataSet* mit der Datenbank verbinden, um bestimmte Datensätze von dort abzuholen (hierzu wird das *SelectCommand*-Objekt des *DataAdapter*s eingesetzt).
- Bei abgekoppelter Datenbank die Änderungen direkt im *DataSet* vornehmen (hierzu ist eine SQL-Anweisung leider untauglich, da das *DataSet* kein SQL kennt).
- Das *DataSet* irgendwann mal wieder mit der Datenbank verbinden, um die Inhalte zu aktualisieren (hierzu werden *UpdateCommand*-, *InsertCommand*- und *DeleteCommand*-Objekt des *DataAdapter*s gebraucht).

Wir wollen das am Beispiel der *Artikel*-Tabelle aus der Datenbank *Nordwind.mdb* demonstrieren.

Oberfläche

Neben zwei *Button*s zum Anzeigen und Aktualisieren brauchen wir noch eine *DataGridView*-Komponente (siehe Laufzeitabbildung).

Quellcode (Command-Objekte selbst programmiert)

```
using System.Data.OleDb;

public partial class Form1 : Form
{
```

Die wichtigsten Objekte sollten global verfügbar sein:

```
    private OleDbConnection conn = new OleDbConnection(
                              "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Nordwind.mdb;");
    private OleDbDataAdapter da = null;
    private DataSet ds = null;
    ...
```

Die folgende Methode *getArtikel* liefert ein gefülltes *DataSet* zurück:

```
    public DataSet getArtikel()
    {
```

SelectCommand-Objekt für *DataAdapter* erstellen (geschieht automatisch beim Instanziieren):

```
        string selStr =
        "SELECT ArtikelNr, Artikelname, Einzelpreis, Mindestbestand FROM Artikel ORDER BY
                                                                          Artikelname";
        da = new OleDbDataAdapter(selStr, conn);
```

Die folgende Anweisung sorgt dafür, dass neu hinzugefügte Datensätze sofort einen Primärschlüssel erhalten.

```
        da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
        conn.Open();
        DataSet ds = new DataSet();
        da.Fill(ds, "Artikel");
        conn.Close();
        return ds;
    }
```

Der Methode *setArtikel* wird ein gefülltes *DataSet* per Referenz übergeben. Mit Hilfe eines *CommandBuilder*s werden im Hintergrund für den *DataAdapter* die *UpdateCommand*, *InsertCommand*- und *DeleteCommand*-Objekte erstellt, die für das Zurückschreiben der im *DataSet* vorgenommenen Änderungen in die Datenbank verantwortlich zeichnen.

```
    public void setArtikel(ref DataSet ds)
    {
        OleDbCommandBuilder cb = new OleDbCommandBuilder(da);

        conn.Open();
```

```
        da.Update(ds, "Artikel");
        conn.Close();
    }
```

Anzeigen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        dataGridView1.DataSource = null;
        ds = getArtikel();
        dataGridView1.DataSource = ds;             // DataGridView mit DataSet verbinden
        dataGridView1.DataMember = "Artikel";
        formatDataGridView(dataGridView1);
    }
```

Aktualisieren:

```
    private void button2_Click(object sender, EventArgs e)
    {
```

Nur die Änderungen zurück in die Datenbank schreiben:

```
        DataSet ds1 = ds.GetChanges();
        if (ds1 != null)
        {
            try
            {
                setArtikel(ref ds1);
```

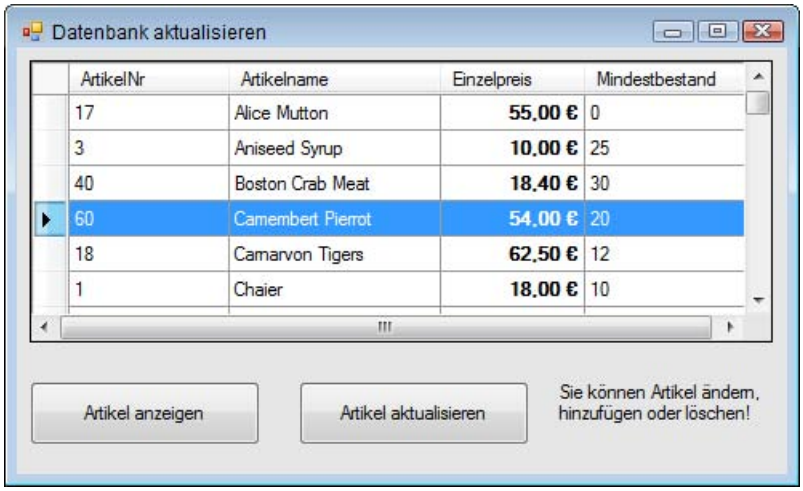
Die per Referenz zurückgegebenen Datensätze werden mit dem Original-*DataSet* zusammengeführt:

```
                ds.Merge(ds1);
                ds.AcceptChanges();
                MessageBox.Show("Datenbank wurde aktualisiert!", "Erfolg");
            }
            catch (Exception ex)
            {
                ds.RejectChanges();
                MessageBox.Show(ex.Message, "Fehler");
            }
        }
    }
}
```

Test

Klicken Sie auf die "Artikel anzeigen"-Schaltfläche, um das *DataSet* anzuzeigen. Nehmen Sie dann einige Änderungen direkt im *DataGridView* vor, fügen Sie Datensätze hinzu (dazu an das Ende des *DataGridView* scrollen) oder löschen Sie Datensätze (mit *Entf*-Taste, vorher komplette Zeile markieren). Klicken Sie auf "Artikel aktualisieren" um die Änderungen in die Datenbank zu

übertragen. Lassen Sie sich dann erneut die Artikel anzeigen um sich davon zu überzeugen, dass alle Änderungen tatsächlich in der Datenbank angekommen sind.



Bemerkungen

- Der Code zur Formatierung der *Einzelpreis*-Spalte des *DataGridView* wurde hier nicht mit abgedruckt (siehe Beispieldaten).
- Es ist auch möglich, mehrere Datensätze hintereinander zu ändern, hinzuzufügen bzw. zu löschen bevor der Abgleich mit der Datenbank erfolgt.
- In der Regel werden Sie nur die von Ihnen selbst hinzugefügten Datensätze löschen können, da die originalen Datensätze in Relationen zu anderen Tabellen eingebunden sind.

Kapitel 24

Das DataSet

Das *DataSet* ist das Kernobjekt von ADO.NET. Da wir im Vorgängerkapitel im Zusammenhang mit der *Fill*- und der *Update*-Methode des *DataAdapter*-Objekts bereits ausführlich auf den Datentransport zwischen Datenquelle und *DataSet* eingegangen sind, können wir uns im vorliegenden Kapitel ganz auf den Aufbau und die spezifischen Funktionalitäten des *DataSet*s konzentrieren.

HINWEIS: Das Problem der Datenbindung an die Benutzerschnittstelle wird hier weitestgehend ausgeklammert und stattdessen in die entsprechenden Spezialkapitel 35 (Windows Forms), 15 (WPF), 43 (Silverlight) und 41 (ASP.NET) ausgelagert.

24.1 Grundlegende Features des DataSets

Beim *DataSet* handelt sich um eine recht komplexe "Minidatenbank", die komplett im Arbeitsspeicher gehalten wird und deren Interaktion mit vorhandenen Datenbanken in der Regel vom *DataAdapter*-Objekt gesteuert wird. Allerdings hinkt der Vergleich mit einer Minidatenbank etwas, denn das *DataSet* kennt keinerlei Datenbankmanager (wie z.B. die Jet-Engine bei Access), es kennt keinen aktuellen Datensatz, keinen Cursor und kein SQL. Genau genommen ist das *DataSet* ein clientseitiger Datencache (Pufferspeicher), der die Änderungen mitprotokolliert.

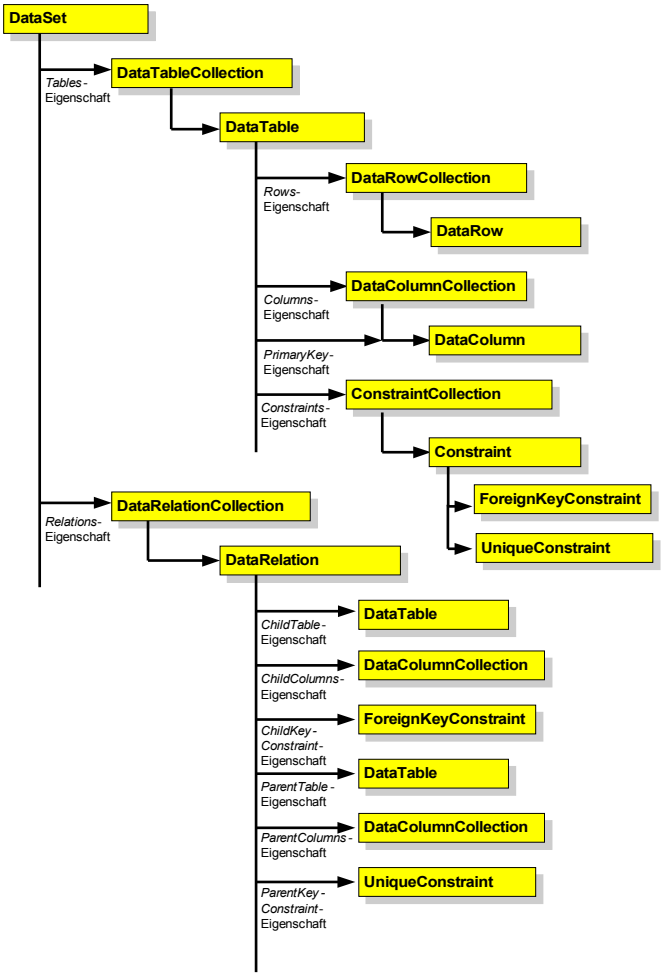
Es spielt auch keine Rolle, von welchem der .NET-Datenprovider das *DataSet* mit Daten gefüllt wurde oder ob es seine Daten auf andere Weise (z.B. aus einer XML-Datei oder direkt aus dem Programm) erhalten hat. Sie werden beim *DataSet*-Objekt auch keine Eigenschaften oder Methoden finden, die Aufschluss über die Datenherkunft geben.

HINWEIS: Dem *DataSet* ist es völlig egal, woher die Daten kommen!

Im Gegensatz zum klassischen Recordset ist das ADO.NET-*DataSet* konsequent objektorientiert aufgebaut.

24.1.1 Die Objekthierarchie

Eine detaillierte Darstellung der Objekthierarchie zeigt die nachfolgende Abbildung.



24.1.2 Die wichtigsten Klassen

Die folgende Tabelle verhilft Ihnen zunächst zu einem groben Überblick über die wichtigsten Klassen aus *System.Data*.

Klasse	Bedeutung
DataSet	Repräsentiert eine "Mini-Datenbank", die (ohne Verbindung zur Datenquelle) autark im Arbeitsspeicher existiert und sowohl Daten (das können mehrere Tabellen sein) als auch Strukturinformationen (Metadaten) und Beziehungen (Relationen) zwischen den Tabellen enthält

Klasse	Bedeutung
DataTable	Entspricht einer einzelnen Tabelle im *DataSet*-Objekt
DataView	Visualisiert eine *DataTable* bzw. einen Ausschnitt davon und erlaubt den Zugriff auf einzelne Zeilen und Spalten
DataRow	Entspricht einer einzelnen Zeile innerhalb eines *DataTable*-Objekts
DataColumn	Entspricht einer einzelnen Spalte (quasi Felddefinition) innerhalb eines *DataTable*-Objekts
DataRelation	Stellt die Verknüpfung zwischen einzelnen Tabellen im *DataSet* her und überwacht die referenzielle Integrität

24.1.3 Erzeugen eines DataSets

In den bisherigen Beispielen des Kapitels haben wir die *DataSet*-Instanzen meist durch Aufrufen eines argumentfreien *DataSet*-Konstruktors erstellt. Dabei wurde die *Name*-Eigenschaft des *DataSet*s automatisch auf *NewDataSet* festgelegt.

Mit einem anderen Konstruktor lässt sich auch ein Namensargument übergeben:

SYNTAX: `DataSet ds = new DataSet(string Name);`

BEISPIEL 24.1: Das Erstellen einer *DataSet*-Instanz.

C#
```
DataSet kuDS = new DataSet("KundenListe");
```

Sie können ein neues *DataSet* aber auch auf Basis eines bereits vorhandenen *DataSet*s erstellen. Beim Kopieren stehen Ihnen folgende Möglichkeiten zur Verfügung:

- Eine exakte Kopie, einschließlich Schema, Daten, Informationen zum Zeilenstatus und Zeilenversionen (siehe *Copy*-Methode).
- Eine Teilmenge des *DataSet*s, die nur die geänderten Zeilen enthält (siehe *GetChanges*-Methode).
- Ein leeres *DataSet*, welches nur das Schema und die relationale Struktur enthält (siehe *Clone*-Methode).

HINWEIS: Diese Möglichkeiten gelten auch für *DataTable*-Objekte!

Copy-Methode

Diese Methode verwenden Sie zum Erstellen einer exakten Kopie des *DataSet*s, die sowohl das Schema (Struktur) als auch die Daten enthält.

BEISPIEL 24.2: Erstellen einer exakten Kopie von *kuDS*.

C#
```
DataSet copyDS = kuDS.Copy();
```

Clone-Methode

Die mit der *Clone*-Methode erzeugte Kopie eines *DataSet*s enthält hingegen nur die Struktur- bzw. Schemainformationen.

BEISPIEL 24.3: Ein Klon eines *DataSet*s wird erstellt.

C#
```
DataSet klonDS = DS.Clone();
```

GetChanges-/HasChanges-Methode

Mit der *GetChanges*-Methode erstellen Sie eine Kopie, die das Schema und nur die Daten enthält, die

- *Added*-Zeilen,
- *Modified*-Zeilen oder
- *Deleted*-Zeilen

darstellen.

Mit *GetChanges* können außerdem nur Zeilen mit einem bestimmten Zeilenstatus geliefert werden, indem beim Aufruf ein *DataRowState*-Wert übergeben wird.

BEISPIEL 24.4: Übergabe eines *DataRowState* beim Aufrufen von *GetChanges*.

C#
```
DataSet changeDS = custDS.GetChanges();                    // alle Änderungen kopieren
DataSet addedDS = custDS.GetChanges(DataRowState.Added);   // nur neue Zeilen kopieren
```

HINWEIS: Durch den Einsatz von *GetChanges* können Sie den Datenverkehr zwischen Geschäftslogik und Datenbank optimieren, weil nicht das gesamte *DataSet* übergeben werden muss, um Änderungen abzugleichen.

Die *HasChanges*-Methode hat eine ähnliche Syntax wie *GetChanges*, nur dass der Rückgabewert *true/false* ist, je nachdem ob Zeilen im *DataSet* geändert wurden oder nicht.

BEISPIEL 24.5: Die folgende Bedingung ist dann erfüllt, wenn zum DataSet *ds* neue Zeilen hinzugefügt wurden. Die neu hinzugefügten Zeilen werden dann in ein weiteres DataSet *ds1* kopiert.

C#
```
if (ds.HasChanges(DataRowState.Added))
   DataSet ds1 = ds.GetChanges(DataRowState.Added);
```

24.2 Das DataTable-Objekt

Die eigentliche Datenspeicherung erfolgt in den *DataTable*-Objekten des *DataSet*s. Es ist deshalb logisch, dass die *DataTable* als wichtigstes und komplexestes Mitglied der *DataSet*-Hierarchie unsere besondere Aufmerksamkeit verdient.

24.2.1 DataTable erzeugen

Bereits im Vorgängerkapitel hatten wir im Zusammenhang mit der *Fill*-Methode des *DataAdapter*s häufig *DataTable*-Objekte erzeugt.

BEISPIEL 24.6: In ein vorhandenes *DataSet* wird mittels eines *DataAdapter*-Objekts *da* ein neues *DataTable*-Objekt *dt* mit der *TableName*-Eigenschaft "Artikel_Liste" eingefügt.

C#
```
da.Fill(ds, "Artikel_Liste");
```

Sie können eine neue *DataTable* auch direkt an die *Tables*-Collection des *DataSet*s anfügen.

BEISPIEL 24.7: Eine *DataTable* wird erzeugt.

C#
```
DataTable dt = ds.Tables.Add("Ku_Liste");
```

24.2.2 Spalten hinzufügen

Das Schema bzw. die Struktur der *DataTable* wird – zusammen mit etwaigen Einschränkungen (Constraints) – durch eine Auflistung von *DataColumn*-Objekten bestimmt (*Columns*-Eigenschaft).

Zum Erstellen von *DataColumn*-Objekten gibt es die beiden .NET-typischen Möglichkeiten

- *DataColumn*-Konstruktor verwenden oder
- *Add*-Methode der *Columns*-Eigenschaft der *DataTable* aufrufen.

Die *Add*-Methode akzeptiert optionale *ColumnName*-, *DataType*- und *Expression*-Argumente oder auch ein vorhandenes *DataColumn*-Objekt. Weil *DataTable*-Objekte nicht spezifisch für einen bestimmten .NET-Datenprovider ausgelegt sind, werden für den Datentyp die .NET Framework-Typen verwendet.

BEISPIEL 24.8: Eine *DataTable* wird erzeugt und vier Spalten werden hinzugefügt. Die Eigenschaften für die *KundenCode*-Spalte verbieten *DBNull*-Werte und verlangen, dass die Werte eindeutig sein müssen.

C#
```
DataTable dt = new DataTable("Kunden");
DataColumn col = dt.Columns.Add("KundenCode", typeof(System.Int32));
col.AllowDBNull = false;
col.Unique = true;
dt.Columns.Add("Firma", typeof(System.String));
dt.Columns.Add("Kontaktperson", typeof(System.String));
dt.Columns.Add("Gehalt", typeof(System.Double));
```

HINWEIS: Geben Sie für eine Spalte keinen Namen an, so erhält sie den inkrementellen Standardnamen "ColumnN...", beginnend mit "Column1", wenn sie zur *DataColumnCollection* hinzugefügt wird.

24.2.3 Zeilen zur DataTable hinzufügen

Haben Sie ein oder mehrere *DataTable*-Objekte erstellt und deren Struktur mit Hilfe von *DataColumn*-, *Constraint*- und *DataRelation*-Objekten definiert, können Sie den Tabellen beliebig viele neue Datenzeilen hinzufügen.

NewRow-Methode

Verwenden Sie die *NewRow*-Methode der *DataTable*, so hat die erzeugte Zeile sofort die zur Tabelle passende Struktur.

SYNTAX: `DataRow myRow = myTable.NewRow();`

Anschließend lässt sich die neu hinzugefügte Zeile mit Hilfe von Spaltennamen oder Index bearbeiten.

BEISPIEL 24.9: Eine neue Zeile wird erzeugt, mit Werten gefüllt und zur *DataTable* hinzugefügt.

C#
```
DataTable dt = ds.Tables["KundenListe"];
DataRow rw = dt.NewRow();
rw["Firma"] = "Catering Service";              // Zugriff über Spaltennamen
rw[2] = "Willy Schneider";                     //   ...   über Index
dt.Rows.Add(rw);                               // Hinzufügen zur DataTable
```

BEISPIEL 24.10: Sieht trickreich aus, aber es funktioniert

C#

Eine neue leere Zeile wird zu einer *DataTable* hinzugefügt.

```
dt.Rows.Add(dt.NewRow());
```

ImportRow-Methode

Um vorhandene Zeilen zu einem *DataTable*-Objekt hinzuzufügen, können Sie die *ImportRow*-Methode der *DataTable* verwenden. Diese übergibt die Daten, den Zeilenstatus sowie Versionsinformationen.

HINWEIS: Spaltenwerte werden nur berücksichtigt, wenn der Spaltenname übereinstimmt und wenn der Datentyp kompatibel ist!

BEISPIEL 24.11: *ImportRow*-Methode

C#

Nachdem ein Klon des *DataSet*s erstellt ist, werden all die Zeilen aus dem ursprünglichen *DataSet* zur *Kunden*-Tabelle im *DataSet*-Klon für Kunden hinzugefügt, deren *Land*-Spalte den Wert "Österreich" hat.

```
DataSet kuGermDS = kuDS.Clone();
DataRow[] cRows = kuDS.Tables["Kunden"].Select("Land = 'Österreich'");
DataTable kuTable = kuGermDS.Tables["Kunden"];
```

BEISPIEL 24.11: *ImportRow*-Methode

```
foreach (DataRow cRow in cRows)
    kuTable.ImportRow(cRow);
```

24.2.4 Auf den Inhalt einer DataTable zugreifen

Eine *DataTable* ist (stark vereinfacht) durchaus mit einem zweidimensionalen Array vergleichbar, wie es die meisten Programmiersprachen kennen. Trotzdem gestaltet sich der Zugriff auf die Array-Elemente etwas komplizierter, denn die gnadenlose Objektorientierung von .NET verlangt, dass anstatt der einfachen Übergabe von Zeilen- und Spaltenindex komplette Objekte (*DataRow*, *DataColumn*) zu überreichen sind.

Columns- und Rows-Eigenschaften

Die beiden wichtigsten Eigenschaften der *DataTable*-Klasse sind die *Columns*- und *Rows*-Auflistungen, weil sie den Zugriff auf Zeilen und Spalten der *DataTable* ermöglichen.

BEISPIEL 24.12: Alle Zeilen und Spalten einer *DataTable* werden innerhalb von zwei geschachtelten *foreach*- Schleifen in eine *ListBox* ausgegeben.

C#
```
DataTable dt = ds.Tables["ArtikelListe"];
foreach (DataRow cRow in dt.Rows)
{
      foreach (DataColumn cCol in dt.Columns)
          listBox1.Items.Add(cCol.ColumnName + " = " + cRow[cCol.Ordinal]);
      listBox1.Items.Add("-----------------------------------------------------");
}
```

HINWEIS: Den vollständigen Quellcode finden Sie im PB 23.8.1 "Wichtige ADO.NET-Objekte im Einsatz".

TableName- und ColumnName-Eigenschaften

Der Name einer Tabelle bzw. Spalte ist über die Eigenschaften *TableName* bzw. *ColumnName* erreichbar. Der Zugriff auf die einzelnen Daten erfolgt über den Indexer, dem entweder der Namen der Spalte, deren fortlaufende Ordinalnummer (beginnend mit 0) oder aber auch eine Instanz der zugehörigen *DataColumn* übergeben werden können.

Normalerweise sind also zwei Schritte auszuführen:

- Das entsprechende *DataRow*-Objekt auswählen und
- über dessen Indexer auf die gewünschte Spalte zugreifen.

BEISPIEL 24.13: Die *Firma* des dritten Kunden eines *DataSet*s soll ausgegeben werden.

C#
```
DataTable dt = ds.Tables["Kunden"];
DataRow rw = dt.Rows[2];                          // Auswahl der Zeile
textBox1.Text = rw[1].ToString();                 // Zugriff auf Firma über Index
```
oder
```
textBox1.Text = rw["Firma"].ToString();           // Zugriff über Spaltennamen
```

Man kann auf eine einzelne Zelle aber auch mit einer einzigen Anweisung zugreifen.

BEISPIEL 24.14: Der äquivalente Code zum Vorgängerbeispiel

C#
```
textBox1.Text = ds.Tables["Kunden"].Rows[2]["Firma"].ToString();
```

Find-Methode

Für das Auffinden einer bestimmten Zeile kann man auch die *Find*-Methode der *Rows*-Auflistung der *DataTable* verwenden. Der Zugriff funktioniert allerdings nur dann, wenn der *DataTable* vorher eine *PrimaryKey*-Eigenschaft zugewiesen wurde.

BEISPIEL 24.15: Der alternative Code zu den Vorgängerbeispielen, wobei aber die Zeilenauswahl über den Primärschlüssel erfolgt (die dritte Zeile hat den *KundenCode* ANTON).

C#
```
DataTable dt = ds.Tables["Kunden"];

DataColumn[] colArr = new DataColumn[1];     // Array mit einem Feld deklarieren
colArr[0] = dt.Columns["KundenCode"];        // Array füllen mit Primärschlüsselspalte
dt.PrimaryKey = colArr;                      // Primärschlüssel zuweisen
DataRow rw = dt.Rows.Find("ANTON");          // Zeilenauswahl über Primärschlüssel
textBox1.Text = rw["Firma"].ToString();      // Spaltenauswahl und Zugriff
```

Zeilen löschen

Es gibt zwei grundsätzlich verschiedene Methoden, um ein *DataRow*-Objekt aus seinem *DataTable*-Objekt zu entfernen:

- Die *Remove*-Methode des *Rows*-Collection-Objekts und
- die *Delete*-Methode des *DataRow*-Objekts.

Remove löscht eine *DataRow* aus der *Rows*-Collection, während *Delete* die Zeile nur zum Löschen markiert.

BEISPIEL 24.16: Die *Remove*-Methode der *Rows*-Collection erhält eine *DataRow* als Argument und entfernt diese aus der Auflistung.

C#
```
dt.Rows.Remove(rw);
```

BEISPIEL 24.17: Die *Delete*-Methode einer *DataRow* wird aufgerufen, um deren *RowState* in *Deleted* zu ändern.

C#
```
rw.Delete;
```

24.3 Die DataView

Das Trennen der Daten von ihrer Darstellung macht es möglich, von einer *DataTable* ganz verschiedene Ansichten zu erstellen, ohne dass die Daten im Speicher dupliziert werden müssen. Die *DataView*-Klasse liefert die Ansicht einer *DataTable* nicht nur zwecks Datenanzeige, sondern erlaubt auch das Filtern, Sortieren und Suchen von Datensätzen.

24.3.1 Erzeugen einer DataView

Dem üblichen Konstruktor wird das zugrunde liegende *DataTable*-Objekt übergeben:

SYNTAX: `DataView dv = new DataView(DataTable dt );`

BEISPIEL 24.18: Zwei verschiedene Sichten der *Artikel*-Tabelle von *Nordwind.mdb* werden erzeugt.

C#
```
OleDbConnection conn = new OleDbConnection(
                          "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
OleDbCommand cmd = new OleDbCommand("SELECT * FROM Artikel", conn);
OleDbDataAdapter da = new OleDbDataAdapter(cmd);
DataSet ds = new DataSet();
conn.Open();
da.Fill(ds, "Artikel_Liste");
conn.Close();
DataTable dt = ds.Tables[0];
DataView dv1 = new DataView(dt);
...
DataView dv2 = new DataView(dt);
...
```

24.3.2 Sortieren und Filtern von Datensätzen

Zum Sortieren übergeben Sie der *Sort*-Eigenschaft einfach die kommaseparierten Spalten, nach denen sortiert werden soll. Anschließend steht "ASC" für eine aufsteigende und "DESC" für eine absteigende Sortierfolge.

Das Filtern von Datensätzen ähnelt der WHERE-Klausel in einer SQL-Abfrage. Die gewünschte Filterbedingung wird der *RowFilter*-Eigenschaft zugewiesen. Die anschließende Sicht auf die *DataTable* enthält nur noch die Datensätze, auf welche die Bedingung zutrifft.

BEISPIEL 24.19: Aus der Tabelle *Artikel* werden alle Artikel mit einem Preis unter 20 Euro herausgefiltert und nach dem Artikelnamen (mit "Z" beginnend) umsortiert.

C#
```
DataTable dt = ds.Tables["Artikel"];
DataView dv = new DataView(dt);
dv.RowFilter = "Artikelname LIKE 'A%' AND Einzelpreis < 20";
dv.Sort = "Artikelname DESC";
```

HINWEIS: Den kompletten Code enthält das PB 24.6.1 "Im DataView sortieren und filtern".

24.3.3 Suchen von Datensätzen

Die beiden Methoden *Find* und *FindRows* erlauben (in Kombination mit einer vorgegebenen Sortierung) das Auffinden von Zeilen, bei denen die Sortierspalten mit den angegebenen Werten übereinstimmen.

Die *Find*-Methode liefert nur den Index der ersten Fundstelle bzw. –1, wenn kein Datensatz gefunden wurde. Hingegen überbringt die *FindRows*-Methode ein *DataRowView*-Array mit allen passenden Zeilen.

BEISPIEL 24.20: In der *Kunden*-Tabelle der *Nordwind*-Datenbank wird der erste Kunde mit der *Kontaktperson* "Hanna Moos" gesucht und in einem *DataGridView* markiert.

C#
```
DataView dv =  ds.DefaultViewManager.CreateDataView(dt);
dv.Sort = "Kontaktperson";            // in dieser Tabellenspalte wird gesucht
int i = dv.Find("Hanna Moos") ;       // Zeile wird gesucht
dataGridView1.CurrentCell = dataGridView1.Rows[i].Cells[0];    // Zeile wird vorn markiert
if (i == -1)
   MessageBox.Show("Keinen Datensatz gefunden!") ;
```

HINWEIS: Ein entsprechendes Beispiel finden Sie unter PB 24.6.2 "Suche nach Datensätzen".

24.4 Typisierte DataSets

Visual Studio verfügt über Assistenten bzw. Designer, mit deren Hilfe man so genannte "Typisierte DataSets" (*Typed DataSets*) erstellen kann. Auf den Einsteiger mögen diese Konstrukte durchaus verwirrend wirken, hat er doch mit den "normalen" *DataSets* bereits genug zu kämpfen.

Ein "typisiertes DataSet″ ist eine von der Klasse *DataSet* abgeleitete Klasse mit Eigenschaften und Methoden, die einen wesentlich bequemeren, weil streng objektorientierten, Zugriff auf den Inhalt ermöglichen, als dies beim konventionellen DataSet der Fall ist.

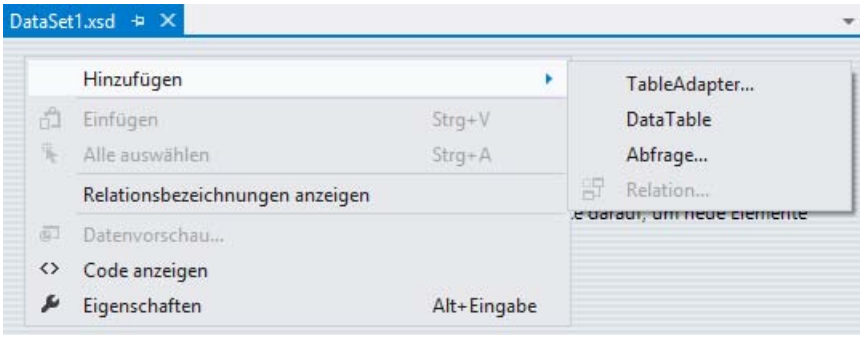
Zu einem typisierten DataSet gehören auch eine XML-Schema-Beschreibung (*.xsd*-Datei) sowie die Beschreibung der Anordnung der Daten in der grafischen Ansicht (*.xss*- und *.xcs*-Dateien).

24.4.1 Ein typisiertes DataSet erzeugen

Dank Assistenten-Unterstützung ist diese Arbeit in wenigen Schritten erledigt, die wir hier nur stichpunktartig aufzählen wollen.

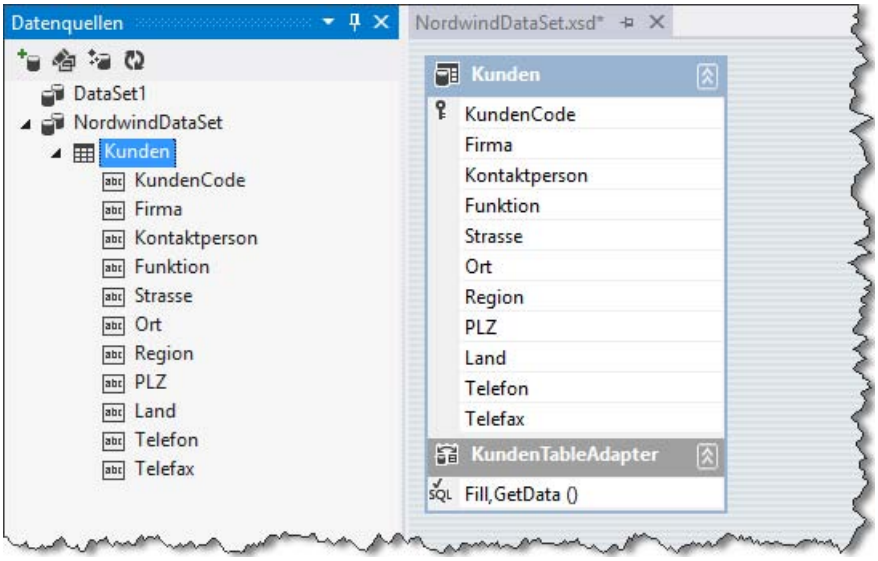
Wählen Sie das Menü *Projekt/Neues Element hinzufügen...* und die Vorlage "DataSet". Es wird eine *.xsd*-Datei (*DataSet1.xsd*) für ein typisiertes DataSet erzeugt.

Anschließend öffnet sich der DataSet-Designer, der z.B. per Drag & Drop aus dem Server-Explorer gefüllt werden kann. Sie können aber auch, wie in der folgenden Abbildung gezeigt, einen *TableAdapter* hinzufügen, um direkt auf eine bestimmte Datenbank zuzugreifen.



Mit dem *TableAdapter-Konfigurations-Assistenten* wählen Sie Ihre Datenverbindung aus und geben im nachfolgenden SQL-Dialog z.B. die Anweisung "SELECT * FROM Kunden" ein.

Wenn der Assistent fertig ist, bieten Ihnen das Datenquellen-Fenster (Menü *Daten/Datenquellen anzeigen*) und der DataSet-Designer den in der folgenden Abbildung gezeigten Anblick.



Das typisierte DataSet kann zur Drag & Drop-Datenbindung in Windows Forms-Fenstern oder per Programmcode verwendet werden.

HINWEIS: Beachten Sie, dass es sich beim typisierten DataSet um eine Klasse handelt, die vor ihrer Verwendung erst noch zu instanziieren ist!

BEISPIEL 24.21: Das typisierte DataSet *DataSet1* (siehe obige Abbildung) wird instanziiert, mit Daten gefüllt und angezeigt.

C#
```
using DataSet1TableAdapters;                          // Namespace für TableAdapter
KundenTableAdapter kta = new KundenTableAdapter();    // TableAdapter instanziieren
DataSet1 ds1 = new DataSet1();                        // typ. DataSet instanziieren
kta.Fill(ds1.Kunden);                                 // typ. DataSet füllen
dataGridView1.DataSource = ds1.Kunden;                // ... und anzeigen
```

24.4.2 Das Konzept der Datenquellen

Wie Sie bereits erfahren haben, sind typisierte DataSets und Datenquellen eng miteinander verknüpft. Ganz allgemein repräsentiert eine Datenquelle die Daten, die für die Applikation verfügbar sind. Die Daten müssen aber nicht unbedingt von einer Datenbank kommen. Der *Assistent zum Konfigurieren von Datenquellen* erlaubt Daten aus drei verschiedenen Quellen:

- **Datenbank**
  Das kann entweder eine serverbasierte Datenbank wie der SQL Server oder Oracle sein, oder eine filebasierte Datenbank wie Access. Visual Studio generiert automatisch typisierte DataSets und andere Klassen und fügt diese zu Ihrem Projekt hinzu.
- **Objekt**
  Irgendein Objekt mit öffentlichen Eigenschaften kann ebenfalls als Datenquelle dienen. Es ist nicht notwendig, dafür ein spezielles Interface zu implementieren.
- **Webservice**
  Beim Erzeugen einer Datenquelle von einem Webdienst werden Objekte erstellt, die mit dem Datentyp korrespondieren, der vom Webdienst geliefert wird.

Eine Datenquelle dient einem doppelten Zweck: Zum Ersten ist es ein Weg um das Erstellen streng typisierter Klassen zu spezifizieren, welche die Anwendungsdaten repräsentieren, zum Zweiten stellt die Datenquelle einen flexiblen Mechanismus zur schnellen Entwicklung attraktiver und funktioneller WinForms und WebForms Benutzerschnittstellen bereit.

HINWEIS: Eine schrittweise Anleitung zum Einrichten einer Datenquelle unter Windows Forms bietet Ihnen das PB 26.8.1 "Einrichten und verwenden einer Datenquelle".

24.4.3 Typisierte DataSets und TableAdapter

Eine Datenbank-Datenquelle ist die Kombination eines streng typisierten *DataSets* mit einem oder mehreren Pärchen von streng typisierten *DataTable*s und *TableAdapter*s.

Zusätzlich werden für jede *DataTable* des *DataSet*s drei abgeleitete Klassen generiert: eine typisierte *DataTable*, eine typisierte *DataRow* und eine typisierte *DataRowChangeEvent* Klasse.

BEISPIEL 24.22: Eine Datenquelle, basierend auf der *Customers* Tabelle der *Northwind*-Datenbank, führt zum Generieren folgender typisierter Klassen:

C#
```
NorthwindDataSet
CustomersDataTable
CustomersDataRow
CustomersRowChangeEvent
CustomersTableAdapter
```

Bereits die ersten vier Klassen des Beispiels charakterisieren ein typisiertes *DataSet*. In Visual Studio wird aber noch eine fünfte Klasse generiert, ein typisierter *TableAdapter* mit dem Namen *CustomersTableAdapter*, auf den wir noch ausführlicher zu sprechen kommen werden.

Warum sollte man ein typisiertes DataSet einsetzen?

Hier die wichtigsten Vorzüge im Detail:

- *DataSet*s, *DataTable*s, *DataRow*s und *RowChangeEvent* sind vom Datenschema abhängig.
- *Tables*, *Columns* und *Relations* stehen als benannte Eigenschaften zur Verfügung und nicht mehr als allgemeine Mitglieder einer Auflistung.
- Der Programmierer hat unter Visual Studio 2012 die volle Unterstützung der IntelliSense (automatische Codevervollständigung), was die Entwicklung des Codes beschleunigt und die Wahrscheinlichkeit von Tippfehlern verringert.
- Fehlerprüfungen erfolgen bereits beim Kompilieren (z.B. führt ein falscher Feldname bereits zu einem Kompilier- und nicht erst zu einem Laufzeitfehler).
- Der Code ist konsistenter und lesbarer (siehe folgendes Beispiel).

BEISPIEL 24.23: Vergleich der Schreibweisen beim Zugriff auf die *ContactName*-Spalte in der vierten Zeile der *Customers*-Tabelle der *Northwind*-Datenbank.

C#

Normales DataSet:

```
country = ds.Tables["Customers"].Rows[3]["ContactName"];
```

Typisiertes DataSet:

```
country = northwindDataSet.Customers[3].ContactName;
```

Die zweite Variante ist doch viel transparenter – oder? Die von seinem untypisierten Vorfahren geerbte Funktionalität bleibt natürlich erhalten, Sie können deshalb ein typisiertes DataSet optional auch mit der gleichen Syntax wie ein normales DataSet abfragen.

Der TableAdapter

Ein *TableAdapter* ist das streng typisierte Äquivalent zum normalen *DataAdapter*. Sie verwenden den *TableAdapter* zunächst genauso wie den *DataAdapter*, d h. zur Verbindungsaufnahme mit einer Datenbank, zum Ausführen von Abfragen (oder gespeicherten Prozeduren) und zum Befüllen einer *DataTable* mit Daten.

BEISPIEL 24.24: Die (typisierte) Tabelle *northwindDataSet.Customers* wird mit Daten gefüllt.

C#
```
this.customersTableAdapter.Fill(this.northwindDataSet.Customers);
```

Mit dem *TableAdapter* von Visual Studio muss man nach Definition einer *Fill*...-Abfragemethode lediglich eine einzige Zeile Code schreiben um einen (oder auch mehrere) Parameterwerte zu übergeben:

BEISPIEL 24.25: Der Abfragemethode *FillByCity* wird als Parameter der Name einer Stadt aus einer *TextBox* übergeben.

C#
```
this.customersTableAdapter.FillByCity (this.northwindDataSet.Customers, this.textBox1.Text);
```

Da die Abfragemethoden streng typisiert sind, erhalten Sie beim Schreiben von Code die volle Unterstützung der Intellisense von Visual Studio.

HINWEIS: Eine Anleitung, wie Sie zu einem *TableAdapter* Abfragemethoden hinzufügen, erhalten Sie im PB 36.8.1 "Einrichten und verwenden einer Datenquelle".

24.5 Die Qual der Wahl

DataSet, DataReader oder LINQ? Die Frage, für welche Datenzugriffstechnologie er sich denn nun entscheiden, soll, bereitet angesichts der Fülle von Objekten manchem .NET-Entwickler schlaflose Nächte.

In diesem zusammenfassenden Abschnitt wollen wir das Thema ADO.NET abrunden und das Für und Wider von DataSet und *DataReader*, sowie die Rolle der neueren ORM[1]-Tools (LINQ to SQL, LINQ to Entities) diskutieren.

[1] Objektrelationale Mapper

24.5.1 DataReader – der schnelle Lesezugriff

Am schnellsten geht das Lesen von Daten mit dem *DataReader* (siehe Kapitel 23), dieser ist in der .NET-Klassenbibliothek das grundlegende Instrument für den Zugriff auf Datenbanken. Alle anderen Datenzugriffstechnologien greifen hinter den Kulissen auf den *DataReader* zu.

Allerdings kann der *DataReader* die Daten nur lesen und die Ergebnismenge sequenziell von vorn nach hinten durchlaufen. Während seiner Arbeit braucht der *DataReader* eine kontinuierliche Verbindung zur Datenbank, deshalb lässt er sich nicht serialisieren und in einem verteilten System nutzen.

Wie schnell der *DataReader* ist, hängt vom Datenbanktreiber, der Datenquelle und der Netzwerkverbindung ab. Von Hause aus sind im .NET-Framework nur ADO.NET-Treiber für den SQL Server und für Oracle enthalten. Weitere Datenbanktreiber werden direkt von den Herstellern oder von Drittanbietern geliefert[1].

Der DataReader ist zwar schnell, kann aber leider nur lesen. Will man Daten speichern, so braucht der *DataReader* einen "schreibfähigen" Partner. In der Regel wird man hierfür *Command*-Objekte einsetzen, mit denen die entsprechenden SQL-Befehle abgesetzt oder gespeicherten Prozeduren aufgerufen werden (siehe Abschnitt 23.3.1 oder PB 23.8.2 "Eine Aktionsabfrage ausführen"). Für den Programmierer bleibt also viel mühselige Kleinarbeit übrig. Dafür wird er mit der besten Performance und dem geringsten Ressourcen-Verbrauch belohnt.

24.5.2 DataSet – die Datenbank im Hauptspeicher

Gegenüber dem *DataReader* ist das *DataSet* ein Alleskönner. Es ist ein im Hauptspeicher gehaltener Datencontainer, der mehrere Ergebnismengen gleichzeitig aufnehmen und miteinander verknüpfen kann. Wie bei einer "richtigen" Datenbank kann man zwischen Datensätzen navigieren und Filter- und Suchoperationen durchführen. Vor allem aber ist der Entwickler in der Lage, Daten im *DataSet* zu ändern. Das *DataSet* unterstützt den Programmierer beim Zurückschreiben der Änderungen in die Datenbank. Ein *DataSet* kann im XML- oder im Binärformat serialisiert und über das Netz verschickt werden.

Bei so viel Licht gibt es natürlich auch Schatten: Im Unterschied zum *DataReader* verschlingt das *DataSet* Unmengen an Speicherplatz und benötigt eine deutlich höhere Rechnerleistung.

Typisiertes versus untypisiertes DataSet

In das normale *DataSet* können Sie beliebige Tabellenschemen einlesen, es ist also untypisiert (so wie auch der *DataReader*). Diese Flexibilität hat den Nachteil, dass man im Programmcode die Spalten nur als Zeichenketten benennen kann, die vom Compiler nicht überprüfbar sind.

Ein typisiertes DataSet hingegen ist eine automatisch generierte Wrapper-Klasse, welche die *DataSet*-Klasse "umhüllt" und ein definiertes Schema zusammen mit Eigenschaften und Methoden besitzt (siehe Abschnitt 24.4).

---

[1] Wer noch mit Microsoft Access arbeitet, ist nach wie vor auf die relativ langsamen OLEDB-Treiber angewiesen.

Neben dem sauberen objektorientierten Code und der Intellisense-Unterstützung bieten typisierte DataSets auch Vorteile bei der Datenbindung, denn hier erleichtern Assistenten und Auswahlfelder in Visual Studio die Arbeit des Entwicklers deutlich.

Doch dieser Komfort hat seinen Preis: Beim Erzeugen typisierter DataSets ist man in der Regel auf die Unterstützung eines Assistenten angewiesen, dieser produziert im Hintergrund Unmengen von Code. Auch die Performance ist deutlich schlechter als beim einfachen, untypisierten DataSet.

Ein DataSet (egal ob normal oder typisiert) hat eigentlich nur in klassischen Desktop-Anwendungen seinen angestammten Platz. Hier überwiegen die Vorteile und die schlechte Performance fällt kaum ins Gewicht. Anders sieht dies bei Webanwendungen und in verteilten Systemen aus, insbesondere bei größeren Datenmengen und Benutzerzahlen. Hier stößt das DataSet bezüglich Leistung und Skalierbarkeit an seine Grenzen.

HINWEIS: Dies ist sicher auch ein Grund, dass für Silverlight-Anwendungen keine *DataSet*-Unterstützung vorhanden und auch nicht geplant ist.

24.5.3 Objektrelationales Mapping – die Zukunft?

Wer bisher nur mit *DataSet* und *DataReader* gearbeitet hat weiß um die Komplexität einer ausgewachsenen Datenbankanwendung. In der Regel sind nicht nur einfache CRUD[1]-Operationen zu programmieren, sondern auch das Problem der Änderungsverfolgung ist zu lösen. Weitere Hürden lauern, wenn es um Primär- und Fremdschlüssel, Autowertspalten, Relationen und Transaktionen geht.

Ideal wäre ein typisierter Datenzugriff unter Vermeidung der Nachteile des DataSets. Dabei besteht das grundsätzliche Problem im Wesensunterschied zwischen dem relationalen Modell und dem Objektmodell. Relationale Datenbanken werden in Tabellen und Beziehungen normalisiert, während Objekte die Konzepte der Vererbung und Komposition sowie komplexe Referenzhierarchien verwenden. Das verhindert ein direktes Mapping der Tabellen und Records an äquivalente Objekte und Auflistungen.

Die Lösung dieses Konflikts soll das objektrelationale Mapping (ORM) bringen. Die Grundidee besteht in der Definition von .NET-Klassen, deren Aufbau dem Tabellenschema einer Datenbank entspricht.

LINQ to SQL/LINQ to Entities

Nachdem Microsoft recht lange das ORM-Feld Drittanbietern überlassen hat[2], gibt es seit .NET 3.5 *LINQ to SQL* und *LINQ to Entities* bzw. das ADO.NET Entity Framework.

HINWEIS: Einen allerersten Eindruck von LINQ to SQL kann Ihnen das Praxisbeispiel 24.6.5 vermitteln.

[1] Create, Retrieve, Update, Delete

[2] z B. *Hibernate bzw. NHibernate*

Die modernen ORM-Technologien lohnen sich für den Datenbankprogrammierer vor allem dann, wenn er ein völlig neues Projekt realisieren möchte, eine Datenbank also noch nicht vorhanden ist. In diesem Fall kann er mit dem Entwurf des Objektmodells beginnen und die Datenbank auf dieser Grundlage erzeugen.

Auch wenn Sie Wert auf sauberen objektorientierten Code, Vererbungsunterstützung, auf Datenbankunabhängigkeit und auf die Kommunikation mit anderen Plattformen legen, sollten Sie LINQ to SQL oder das leistungsfähigere LINQ to Entities ins Auge fassen.

Andererseits kann es für den geplagten Datenbankprogrammierer auch ein fataler Irrtum sein, in den ORM-Tools eine Allzweck-Wunderwaffe zu sehen. Für Massendatenänderungen sind ORMs kaum geeignet, denn es hat keinen Sinn, tausende Objekte erst in den Speicher zu laden, dort zu ändern und anschließend in die Datenbank zurückzubefördern. Hier sollten Sie gezielt mit gekapselten *Command*-Anweisungen SQL-Befehle direkt an die Datenbank schicken.

Ein weiterer Schwachpunkt ist die Geschwindigkeit. Erwartungsgemäß sind ORMs keinesfalls schneller als typisierte *DataSets* oder gar *DataReader*. Damit fällt objektrelationales Mapping für zeitkritische Lese- und Schreiboperationen unter den Tisch. Nach wie vor sind hier der *DataReader* und das direkte SQL mit *Command*-Objekten zu empfehlen.

24.6 Praxisbeispiele

24.6.1 In der DataView sortieren und filtern

Ein *DataView*-Objekt visualisiert die Daten eines *DataTable*-Objekts und ermöglicht gleichzeitig ein bequemes Suchen und Filtern. Das vorliegende Beispiel zeigt eine einfache Anwendung zum Sortieren und Filtern auf Basis der *Artikel*-Tabelle aus *Nordwind.mdb*.

Oberfläche

Wir brauchen ein *DataGridView*, einen *Button* und zwei *TextBox*en. Letztere dienen der Eingabe der *Sort*- und der *RowFilter*-Eigenschaft des *DataView*-Objekts. Die Syntax dieser Eigenschaften ist SQL-orientiert. Um beim Experimentieren nicht jedes Mal komplett die *Sort*- und *RowFilter*-Eigenschaften neu eintippen zu müssen, sollten Sie gleich zur Entwurfszeit beiden *TextBox*en gültige Anfangswerte zuweisen, z.B.

- *Sort*: *Artikelname DESC*
- *RowFilter*: *Artikelname LIKE 'F%' AND Einzelpreis < 60*

Quellcode

```
using System.Data.OleDb;

public partial class Form1 : Form
{
    ...
```

Alle benötigten Objekte werden global referenziert:

```
    private OleDbConnection conn = null;
    private OleDbCommand cmd = null;
    private OleDbDataAdapter da = null;
    private DataTable dt = null;

    private DataView dv = null;
```

Beim Laden der Anwendung erfolgt das Instanziieren und Initialisieren der Objekte:

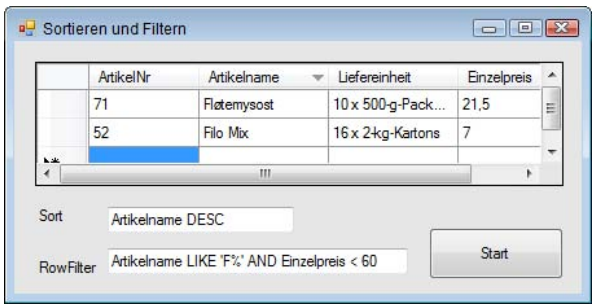
```
    private void Form1_Load(object sender, System.EventArgs e)
    {
        conn = new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;
                 Data Source=Nordwind.mdb;");
        cmd = new OleDbCommand(
                "SELECT ArtikelNr, Artikelname,Liefereinheit,Einzelpreis FROM Artikel", conn);
        da = new OleDbDataAdapter(cmd);
        dt = new DataTable();
        da.Fill(dt);                          // DataTable füllen
        dv = dt.DefaultView;                  // Erzeugen des DataView in Standardansicht
        dataGridView1.DataSource = dv         // Datengitter an DataView anbinden
    }
```

Zum Filtern und Sortieren werden die Inhalte aus den Textboxen zugewiesen:

```
    private void button1_Click(object sender, System.EventArgs e)         // Start
    {
        dv.Sort = textBox1.Text;
        dv.RowFilter = textBox2.Text;
    }
}
```

Test

Nach dem Programmstart zeigt die *DataView* zunächst alle Artikel in Standardansicht an. Klicken Sie auf die *Start*-Schaltfläche, werden z.B. nur noch alle mit "F" beginnenden Artikel mit einem Einzelpreis von z.B. unterhalb *60 Euro* in umgekehrter alphabetischer Reihenfolge angezeigt.



Weiteren Experimenten steht nun nichts mehr im Wege (bescheidene SQL-Kenntnisse vorausgesetzt).

Bemerkungen

- Die SQL-Syntax der *RowFilter*-Eigenschaft ist dieselbe wie die der *Expression*-Eigenschaft des *DataColumn*-Objekts.
- Neben dem "%"-Platzhalterzeichen können Sie auch das Zeichen "*" verwenden.

24.6.2 Suche nach Datensätzen

Die im PB 24.6.1 "Im DataView sortieren und filtern" vorgestellte *RowFilter*-Eigenschaft ermöglicht unter Verwendung der SQL-Syntax (LIKE) und des Platzhalterzeichens "%" auch eine komfortable Suche. Dabei muss nicht der komplette Suchbegriff eingegeben werden, sondern die ersten Buchstaben reichen aus, um ähnliche Datensätze herauszufiltern.

Oberfläche

Für das Startformular benötigen Sie lediglich ein *DataGridView*, eine *ComboBox* und eine *TextBox* (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.OleDb;

    public partial class Form1 : Form
    {
        private DataView dv = null;

        private void Form1_Load(object sender, EventArgs e)
        {
            OleDbConnection conn = new OleDbConnection(
                          "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;");
          OleDbCommand cmd = new OleDbCommand("SELECT * FROM Kunden", conn);
          OleDbDataAdapter da = new OleDbDataAdapter(cmd);
          DataTable dt = new DataTable("Kundenliste");
          conn.Open();
          da.Fill(dt);
          conn.Close();
           dv = new DataView(dt);           // oder dv = dt.DefaultView;
```

Die Übertragung der Spaltenbezeichner in die *ComboBox*:

```
         foreach (DataColumn c in dt.Columns)
                    comboBox1.Items.Add(c.ColumnName);
```

Standardanzeige einstellen:

```
           comboBox1.SelectedIndex = 1;               // Spalte "Firma"
```

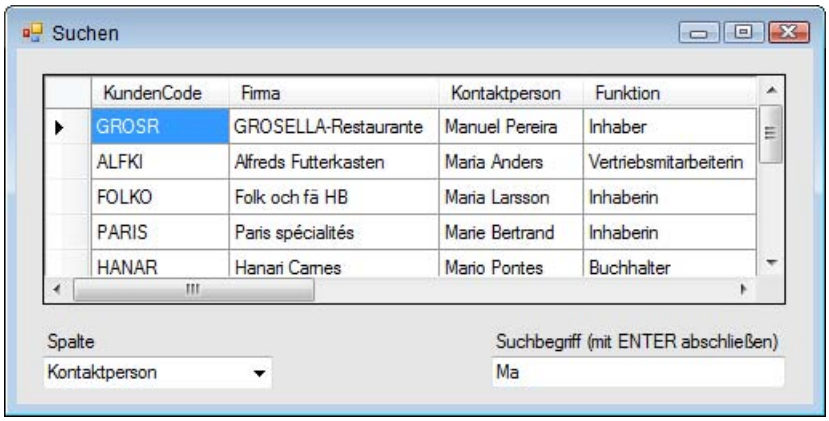
```
            textBox1.Text = "L";                        // Default-Suchbegriff
            dataGridView1.DataSource = dv;
        }
```

Die Suche startet nach Betätigen der *Enter*-Taste:

```
        private void textBox1_KeyUp(object sender, KeyEventArgs e)
        {
            if (e.KeyCode == Keys.Enter)
            {
                dv.Sort = comboBox1.Text;
                dv.RowFilter = dv.Sort + " LIKE '" + textBox1.Text + "%'";
            }
        }
}
```

Test

Stellen Sie in der *ComboBox* zuerst die Spalte ein, die Sie durchsuchen möchten. Geben Sie dann in die *TextBox* ein oder mehrere Zeichen für die Anfangsbuchstaben des zu suchenden Begriffs ein und beenden Sie die Eingabe mit der *Enter*-Taste.



Geben Sie einen leeren Suchbegriff ein, wird wieder die komplette Tabelle angezeigt.

24.6.3 Ein DataSet in einen XML-String serialisieren

Dieses Beispiel zeigt Ihnen nicht nur, wie Sie ein beliebiges *DataSet*-Objekt in einen XML-String verwandeln, sondern erklärt auch den umgekehrten Weg, nämlich die Rücktransformation eines XML-Strings in ein *DataSet*.

Ganz nebenbei wird auch noch demonstriert, wie man ein *DataSet* (inklusive Relationen zwischen den Tabellen) in "Handarbeit" – also ganz ohne Datenbank – erstellen und füllen kann.

Konvertierungsmethoden

Grundlage beider Konvertierungen sind Überladungen der *WriteXml*- bzw. *ReadXml*-Methode des *DataSet*s, welche diesmal nicht auf die Festplatte, sondern direkt auf den Arbeitsspeicher zugreifen.

```
using System.Xml;
using System.IO;
```

Die folgende Methode konvertiert das übergebene *DataSet* in einen XML-String, wobei der Weg über einen *MemoryStream* und ein *Byte*-Array geht:

```
public string ConvertDataSetToXML(DataSet ds)
{
    MemoryStream stream = null;
    XmlTextWriter writer = null;
    try
    {
        stream = new MemoryStream();
```

XmlTextWriter mit dem *MemoryStream* initialiseren:

```
        writer = new  XmlTextWriter(stream, Encoding.Unicode);
```

DataSet in den *MemoryStream* schreiben und dabei auch die Strukturinformationen mit übergeben:

```
        ds.WriteXml(writer, XmlWriteMode.WriteSchema);
```

Byte-Array als Puffer erstellen (*MemoryStream* kann grundsätzlich nur in ein Byte-Array einlesen):

```
        byte[] arr = stream.ToArray();
```

XML-String aus Byte-Array gewinnen und zurückgeben:

```
        UnicodeEncoding utf = new UnicodeEncoding();
        return utf.GetString(arr).Trim();
    }
    catch
    {
        return String.Empty;
    }
    finally
    {
        if (writer != null) writer.Close();
    }
}
```

Die zweite Methode arbeitet in umgekehrter Richtung, sie konvertiert einen übergebenen XML-String in ein *DataSet*, was dank *StringReader*-Objekt auf direktem Weg geht:

```
public DataSet ConvertXMLToDataSet(string xml)
{
```

```
            StringReader reader = null;
            try
            {
                DataSet ds = new DataSet();
                reader = new StringReader(xml);
```

XML-String in *DataSet* einlesen:

```
                ds.ReadXml(reader);
                return ds;
            }
            catch
            {
                return null;
            }
            finally
            {
                if (reader != null) reader.Close();
            }
        }
```

Testoberfläche Form1

Eine *TextBox* (*MultiLine = True*), zwei *Button*s und das "gute alte" *DataGrid* bilden die Testoberfläche. Da man im Unterschied zum modernen *DataGridView* im *DataGrid* auch mehrere Tabellen und ihre Beziehungen gleichzeitig darstellen kann, wurde letzteres extra zu diesem Zweck aus seiner Schmollecke zurückgeholt (Hinzufügen zur Toolbox über Kontextmenü *Elemente auswählen ...*).

Quellcode Form1

```
using System.Xml;
using System.IO;

public partial class Form1 : Form
{
        ...
```

DataSet => XML-String:

```
        private void button1_Click(object sender, EventArgs e)
        {
           DataSet ds = getTestDS();
           textBox1.Text = ConvertDataSetToXML(ds);
           button2.Enabled = true;
        }
```

XML-String => DataSet:

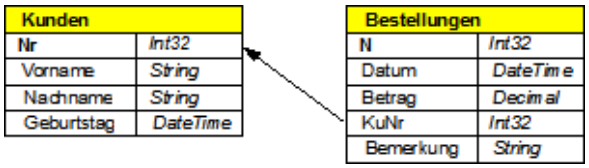
```
        private void button2_Click(object sender, EventArgs e)
        {
```

```
            DataSet ds = ConvertXMLToDataSet(textBox1.Text);
            dataGrid1.DataSource = null;
            dataGrid1.DataSource = ds;
        }
}
```

Erzeugen eines untypisierten DataSets als Testobjekt

Ein *DataSet* zum Experimentieren hätten wir uns viel einfacher aus einer beliebigen Datenbanktabelle holen können (z.B. mittels *Fill*-Methode des *DataAdapter*s). Da es hier aber um grundsätzliche Untersuchungen geht, wollen wir diesmal unser *DataSet* lieber eigenhändig per Code erstellen.

Vorbild ist die abgebildete Struktur, in welcher die Tabellen *Kunden* und *Bestellungen* über eine 1 n-Relation miteinander verbunden sind:



Die Methode *getTestDS* erzeugt ein untypisiertes *DataSet* mit zwei *DataTable*s und einer *DataRelation* entsprechend obiger Abbildung und fügt jeder Tabelle zwei Datensätze hinzu:

```
private static DataSet getTestDS()
{
```

Tabelle "Kunden":

```
    DataTable dt1 = new DataTable("Kunden");
```

Primärschlüssel:

```
    DataColumn col1 = dt1.Columns.Add("Nr", typeof(System.Int32));
    col1.AllowDBNull = false;
    col1.Unique = true;
    col1.AutoIncrement = true;
    col1.AutoIncrementStep = 1;
```

Die restlichen Spalten hinzufügen:

```
    dt1.Columns.Add("Vorname", typeof(System.String));
    dt1.Columns.Add("Nachname", typeof(System.String));
    dt1.Columns.Add("Geburtstag", typeof(System.DateTime));
```

Zwei Datensätze hinzufügen:

```
    DataRow rw11 = dt1.NewRow();
    rw11["Vorname"] = "Klaus";
    rw11["Nachname"] = "Müller";
    rw11["Geburtstag"] = Convert.ToDateTime("3.4.1975");
```

```
            DataRow rw12 = dt1.NewRow();
            rw12["Vorname"] = "Tobalt";
            rw12["Nachname"] = "Tonne";
            rw12["Geburtstag"] = Convert.ToDateTime("5.8.1984");
            dt1.Rows.Add(rw11);
            dt1.Rows.Add(rw12);
```

Tabelle "Bestellungen":

```
            DataTable dt2 = new DataTable("Bestellungen");
            DataColumn col2 = dt2.Columns.Add("Nr", typeof(System.Int32));
            col2.AllowDBNull = false;
            col2.Unique = true;
            col2.AutoIncrement = true;
            col2.AutoIncrementStep = 1;
            dt2.Columns.Add("Datum", typeof(System.DateTime));
            dt2.Columns.Add("Betrag", typeof(System.Decimal));
            dt2.Columns.Add("KuNr", typeof(System.Int32));          // Fremdschlüssel
            dt2.Columns.Add("Bemerkung", typeof(System.String));
```

Zwei Datensätze hinzufügen:

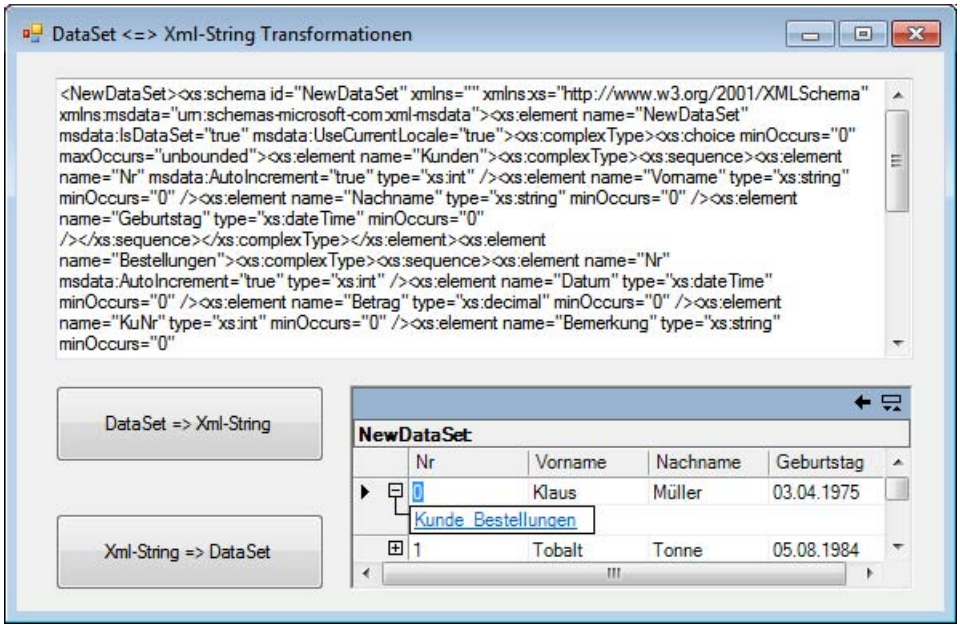
```
            DataRow rw21 = dt2.NewRow();
            rw21["Datum"] = Convert.ToDateTime("20.2.2012");
            rw21["Betrag"] = Convert.ToDecimal("256,50");
            rw21["KuNr"] = 0;
            rw21["Bemerkung"] = "per Nachname";
            dt2.Rows.Add(rw21);

            DataRow rw22 = dt2.NewRow();
            rw22["Datum"] = Convert.ToDateTime("8.3.2012");
            rw22["Betrag"] = Convert.ToDecimal("12,95");
            rw22["KuNr"] = 0;
            rw22["Bemerkung"] = "per Scheck";
            dt2.Rows.Add(rw22);
```

DataSet zusammenbauen (mit 1 : n Relation zwischen *Kunden* und *Bestellungen*):

```
            DataSet ds = new DataSet();
            ds.Tables.Add(dt1);
            ds.Tables.Add(dt2);
            ds.Relations.Add("Kunde_Bestellungen", ds.Tables["Kunden"].Columns["Nr"],
                                                   ds.Tables["Bestellungen"].Columns["KuNr"]);
            return ds;
        }
```

Test

Zunächst lassen wir uns die XML-Darstellung des *DataSet*s in der *TextBox* anzeigen. Anschließend betätigen wir zwecks Rückkonvertierung die untere Schaltfläche:



24.6.4 Untypisiertes in ein typisiertes DataSet konvertieren

Als Ergebnis einer Datenbankabfrage oder eines Webmethodenaufrufs liegt häufig ein "normales" *DataSet* vor, für die weitere Informationsverarbeitung möchte man aber gern ein *typisiertes DataSet* nehmen, welches z.B. als Datenquelle für ein Windows- oder WPF-Frontend agieren soll.

Während der umgekehrte Weg (typisiert → untypisiert) ziemlich einfach ist:

```
DataSet ds = (DataSet) ds1;
```

... funktioniert der folgende Code leider nicht:

```
DataSet1 ds1 = (DataSet1) ds;
```

Der notorische "Dünnbrettbohrer" könnte allerdings mit folgendem Code sein Ziel erreichen:

```
ds.WriteXml("Temp.dat", XmlWriteMode.WriteSchema);
DataSet1 ds1 = new DataSet1();
ds1.ReadXml("Temp.dat");
```

Neben der relativen Langsamkeit hat dieses Verfahren den gravierenden Nachteil, dass als Zwischenspeicher eine temporäre Datei auf der Festplatte herhalten muss. Dies kann z.B. beim Ausführen des Codes auf einem Internetserver mangels Schreibrechten zur Funktionsunfähigkeit führen.

Das vorliegende Beispiel zeigt eine Lösung, wie man die Informationen aus dem gefüllten untypisierten *DataSet*-Objekt *ds* in das leere typisierte *DataSet1*-Objekt *ds1* schaffen kann, ohne dabei mühselig durch alle Zeilen und Spalten der *DataTable*s iterieren zu müssen.

Konvertierungscode

Die folgende Methode konvertiert ein übergebenes untypisiertes *DataSet* in ein DataSet vom Typ *DataSet1*. Im Zentrum stehen dabei die bekannten Methoden *WriteXml* und *ReadXml*, allerdings arbeiten die Methoden diesmal nicht mit einer Datei sondern mit einem *MemoryStream*.

```
public DataSet1 ConvertUntypedToTypedDS(DataSet ds)
{
    MemoryStream stream = null;
    try
    {
        stream = new MemoryStream();
```

DataSet inkl. Strukturinfo in den *MemoryStream* schreiben:

```
        ds.WriteXml(stream, XmlWriteMode.WriteSchema);
```

Position im *MemoryStream* auf Anfang zurücksetzen:

```
        stream.Seek(0, SeekOrigin.Begin);
```

Typisiertes *DataSet* instanziieren und Inhalt des *MemoryStreams* einlesen:

```
        DataSet1 ds1 = new DataSet1();
        ds1.ReadXml(stream, XmlReadMode.InferSchema);
        return ds1;
    }
    catch
    {
        return null;
    }
    finally
    {
        if (stream != null) stream.Close();
    }
}
```

Um die Methode zu testen, brauchen wir sowohl ein gefülltes untypisiertes als auch ein leeres typisiertes DataSet mit identischen Strukturen. Unser Beispiel soll aber nicht nur den Trivialfall einer einzigen Tabelle abdecken, sondern zumindest aus zwei über eine Relation verknüpften Tabellen bestehen.

Normales DataSet erzeugen

Die beiden *DataTable*-Objekte *Kunden* und *Bestellungen* sind über eine 1:n-Relation miteinander verknüpft (ein Kunde hat keine, eine oder mehrere Bestellungen). Der Fremdschlüssel *KuNr* aus der Tabelle *Bestellungen* zeigt auf den Primärschlüssel *Nr* der Tabelle *Kunden*.

Um dieses *DataSet* per Code zu erzeugen, verwenden wir die Methode *getTestDS* aus dem Praxisbeispiel 24.6.3 "Ein DataSet in einen XML-String serialisieren".

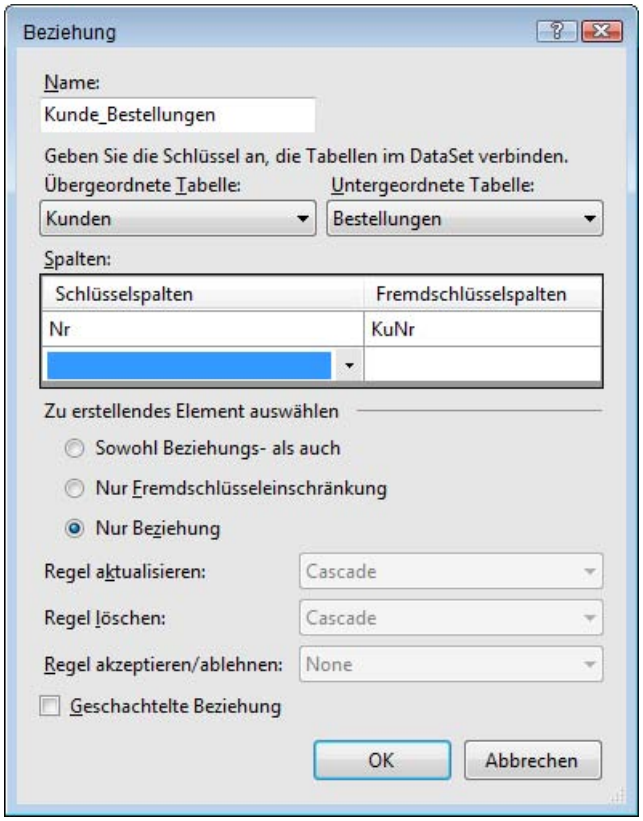
Typisiertes DataSet erzeugen

Ziel ist der Entwurf eines typisierten DataSets gleicher Struktur wie das untypisierte DataSet.

Über das Menü *Projekt/Neues Element hinzufügen...* fügen Sie ein (typisiertes) DataSet hinzu.

HINWEIS: Ein über das Projektmenü hinzugefügtes DataSet ist immer typisiert und automatisch eine Datenquelle!

Mit Hilfe des Menüs *Daten/Datenquellen anzeigen* bringen Sie das "Datenquellen"-Fenster zur Ansicht und entdecken das neu erzeugte typisierte *DataSet1*. Wählen Sie das Kontextmenü *DataSet mit Designer bearbeiten*. Klicken Sie mit der rechten Maustaste auf die leere Oberfläche des Designers und erzeugen Sie über das Kontextmenü *Hinzufügen/DataTable* die Tabellen *Kunden* und *Bestellungen* mit den entsprechenden Spalten. Über das Kontextmenü *Eigenschaften* weisen Sie jeder Spalte den Datentyp zu (vorher die volle Spalte markieren!).



Nachdem auch die Tabelle *Bestellungen* fertig ist, wählen Sie im Kontextmenü *Hinzufügen/Relation...* und verbinden im Dialogfenster "Beziehung" beide Tabellen entsprechend der Abbildung.

Das Ergebnis im DataSet-Designer:

Das Datenquellen-Fenster sollte nun folgenden Anblick bieten:

Oberfläche

Auf dem Startformular *Form1* findet (neben einem *Button*) das *DataGrid* seinen Platz. Gegenüber seinem strahlenden Nachfolger, dem *DataGridView*, hat dieses zum Schattendasein verdammte Control den Vorteil, dass man mit ihm sehr bequem mehrere Tabellen und deren Verknüpfungen betrachten kann.

Der Code beschränkt sich im Wesentlichen auf den Aufruf der Methoden *getTestDS* und *Convert-UntypedToTypedDS*:

```
using System.IO;
public partial class Form1 : Form
{
...
        private void button1_Click(object sender, EventArgs e)
        {
```

Das gefülltes untypisierte DataSet abholen:

```
DataSet ds = getTestDS();
```

Kontrollanzeige (umständlich weil nicht typisiert):

```
MessageBox.Show(ds.Tables["Kunden"].Rows[0]["Vorname"].ToString());
```

Typisiertes DataSet füllen:

```
DataSet1 ds1 = ConvertUntypedToTypedDS(ds);
```

Datengitter mit dem typisierten DataSet verbinden:

```
dataGrid1.DataSource = ds1;
```

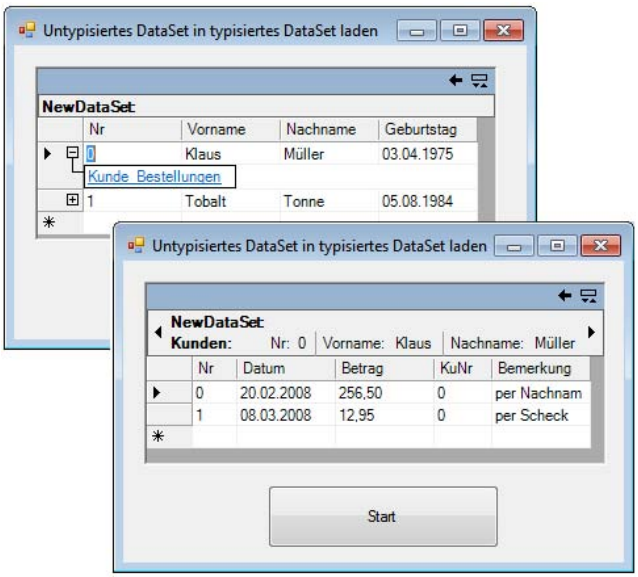
Kontrollanzeige (transparent weil typisiert):

```
MessageBox.Show(ds1.Kunden[0].Vorname);
}
```

Beide Meldungsfenster zeigen das gleiche Ergebnis ("Klaus") und dienen lediglich dem Zweck, die unterschiedliche Syntax beim Zugriff auf untypisiertes und typisiertes DataSet gegenüberzustellen.

Test

Der Programmtest beweist, dass die Methode *ConvertUntypedToTypedDS* unsere Erwartungen voll erfüllt. Die zu einem Kunden gehörigen Bestellungen können über das "+"-Symbol und den Link *Kunde_Bestellungen* aufgeklappt werden.



HINWEIS: Der Wechsel zur jeweils anderen Ansicht ist durch Klick auf den kleinen schwarzen Pfeil rechts oben in der Titelleiste des *DataGrid*s möglich.

Bemerkungen

- Während der untypisierte Zugriff doch ziemlich umständlich ist, bietet der typisierte Zugriff zur Entwurfszeit bessere Transparenz und bequeme Intellisense-Unterstützung.
- Da das klassische *DataGrid*-Control standardmäßig nicht im Werkzeugkasten angezeigt wird, müssen Sie es über das Werkzeugkasten-Kontextmenü *Elemente auswählen ...* und die Registerseite *.NET Framework-Komponenten* herbeiholen[1].

24.6.5 Eine LINQ to SQL-Abfrage ausführen

Mit LINQ to SQL steht Ihnen eine weitere Technologie für den schnellen und einfachen Zugriff auf die Daten des Microsoft SQL Servers zur Verfügung.

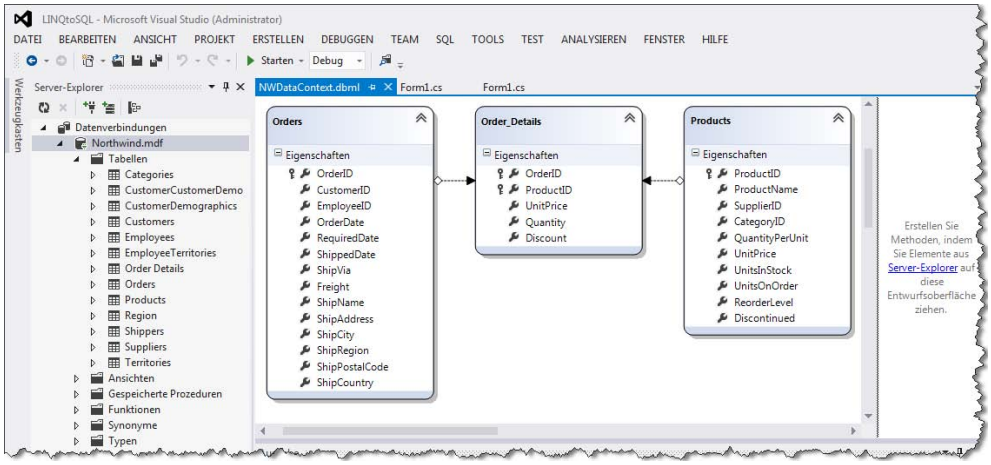
Wir wollen das Prinzip von LINQ to SQL anhand folgender Aufgabenstellung erläutern:

Basierend auf den 1:n-Beziehungen zwischen den Tabellen *Order, Order_Detail* und *Product* in der *Northwind*-Datenbank sind für eine Bestellung die zugehörigen Artikel anzuzeigen.

Datenmodell per LINQ to SQL-Designer erzeugen

Fügen Sie Ihrem Projekt eine neue "LINQ to SQL Klasse" hinzu, um den LINQ to SQL-Designer zu öffnen (*Projekt|Neues Element hinzufügen*). Damit haben Sie bereits die zentrale *DataContext*-Klasse erstellt. Den Namen dieser Klasse können Sie jetzt gegebenenfalls über das Eigenschaftenfenster anpassen (wir wählen *NWDataContext*).

In die noch leere Arbeitsfläche (diese ähnelt dem Klassendesigner) fügen Sie die benötigten SQL-Server-Tabellen ein. Nutzen Sie dazu den Server-Explorer (siehe folgende Abbildung linke Seite).



[1] Es kann einige Zeit dauern, bis der Dialog *Werkzeugkastenelemente auswählen* erscheint.

HINWEIS: Für unser Beispiel fügen Sie die Tabellen *Order, Order_Detail* und *Product* ein.

Der Designer erstellt nachfolgend automatisch die erforderlichen C#-Mapperklassen für die einzelnen Tabellen sowie deren Verknüpfungen.

HINWEIS: Sie können neben reinen Tabellen auch Views oder gespeicherte Prozeduren in den Designer einfügen. Views werden wie Tabellen behandelt, gespeicherte Prozeduren werden als Methoden der *DataContext*-Klasse mit typisierten Rückgabewerten gemappt.

Datenquelle hinzufügen

Möchten Sie Ihre Formulare mit Datenbindung ausstatten, von den Vorteilen eines Datenbanknavigators profitieren und schon zur Entwurfszeit die Darstellung von Tabellen etc. anpassen, kommen Sie nicht um die Definition einer Datenquelle herum[1].

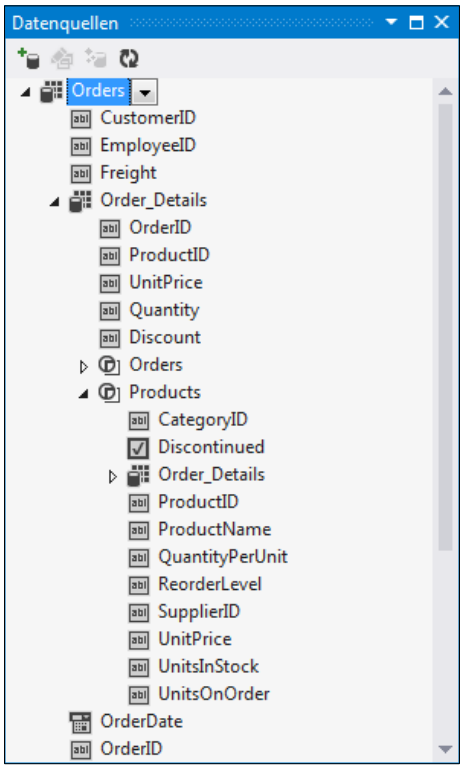
Öffnen Sie das Fenster "Datenquellen" (Menü *Ansicht|Weitere Fenster|Datenquellen anzeigen*) und klicken Sie den Link *Neue Datenquelle hinzufügen...*.

Im "Assistent zum Konfigurieren von Datenquellen" wählen Sie zunächst den Datenquellentyp *Objekt* aus, nachfolgend spezifizieren Sie die Klasse *Orders*.

Nach dem Klick auf "Fertigstellen" erscheint die Objekt-Datenquelle *Orders* im Datenquellenfenster. Um die Klassen *Order_Details* und *Products* brauchen wir uns nicht zu kümmern, diese

[1] Es sei denn, Sie wollen Unmengen von Quellcode schreiben.

wird dank Assoziation bereits unterhalb der Klasse *Orders* in der Liste der Datenquellen aufgeführt:



Oberfläche

Ziehen Sie die Datenquelle *Orders* per Drag & Drop auf das Startformular *Form1,* so wird von einem im Hintergrund agierenden Assistenten automatisch eine Benutzerschnittstelle generiert, die standardmäßig aus *DataGridView*, *BindingSource* und *BindingNavigator* besteht.

Zu diesem Zeitpunkt werden bereits alle Spalten im *DataGridView* angezeigt, Sie können bei Bedarf diese an Ihre Vorgaben anpassen (Kontextmenü *Spalten bearbeiten*).

HINWEIS: Wenn Sie das Programm jetzt starten, bleibt das Datengitter leer, da der *BindingSource* noch keine **Instanz** der *Orders*-Liste zugewiesen wurde.

Ergänzen Sie also den Formularcode wie folgt:

```
public partial class Form1 : Form
{
```

Instanz des *DataContext* erzeugen:

```
    NWDataContext dbnw = new NWDataContext();
```

```
        private void Form1_Load(object sender, EventArgs e)
        {
            ordersBindingSource.DataSource = dbnw.Orders;
```

Alternativ können Sie auch mit LINQ to SQL erst die Daten filtern/sortieren etc.:

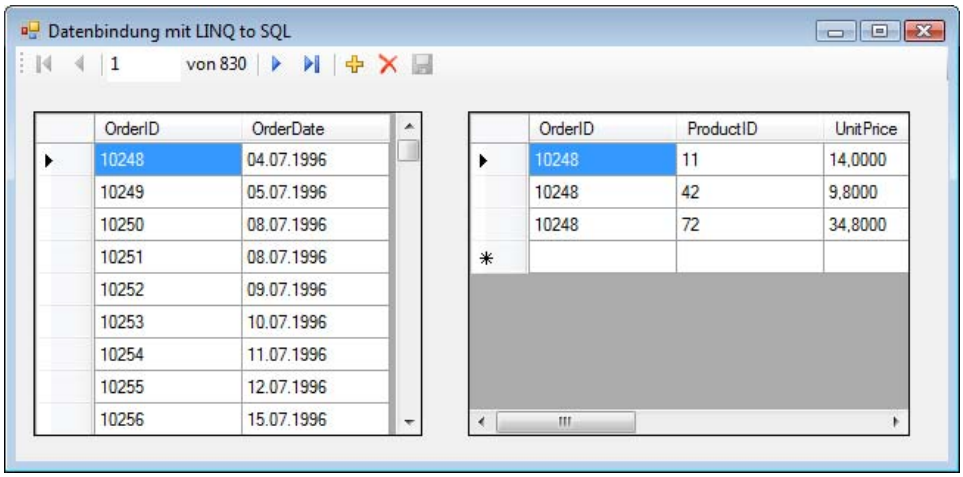
```
            // var orders = from o in dbnw.Orders
            //              ...
            //              select o;
            // ordersBindingSource.DataSource = orders;
        }
```

Auch die Anzeige von Detaildaten, in unserem Fall die den Bestellungen zugeordneten Details, stellt uns vor keine allzu große Hürde. Ziehen Sie einfach aus dem Datenquellenfenster die Klasse *Order_Details* in das Formular, um eine entsprechende *DataGridView* inklusive *BindingSource* zu erzeugen.

Und welchen Code müssen wir schreiben? Keinen, da durch die Assoziation zwischen beiden Klassen bereits alle Informationen vorliegen, um die erforderlichen Daten per SQL abzurufen.

Test

Starten Sie die Anwendung und wechseln Sie im linken *DataGridView* zwischen den einzelnen Bestellungen, um die zugehörigen Details anzuzeigen.



Bemerkung

Ach ja, wie kommen eigentlich neue Datensätze in die Tabellen? Hier genügt es, wenn Sie z.B. ein neues *Order*-Objekt erstellen und es an die *Orders*-Collection anhängen. Mit einem *SubmitChanges* des *DataContext*-Objekts ist die Änderung dann auch schon im Server gelandet.

Kapitel 25

OOP-Spezial

Grundkenntnisse der UML[1] sind für einen Programmierer (egal welcher Sprache) Pflicht.

Dieses Kapitel behandelt dazu zwei Themen:

- eine praktische Einführung in die UML,
- der Klassendesigner von Visual Studio 2012

Beide Beiträge, die relativ lose nebeneinander stehen, sollen dem Programmierer helfen, sich neue Techniken und Tools zu erschließen, welche die UML im Zusammenspiel mit Visual Studio 2012 für die objektorientierte Anwendungsentwicklung bietet.

25.1 Eine kleine Einführung in die UML

Am durchgängigen Beispiel einer Auto-Alarmanlage wollen wir den Umgang mit den wichtigsten UML-Diagrammen erklären.

Folgendes Problem soll modelliert werden:

- Die Zentralverriegelung eines Autos wird mit Hilfe eines Infrarot-Schlüssels betätigt.
- Beim Versuch, den Motor zu starten ohne vorher das Auto mit dem Schlüssel geöffnet zu haben, soll Alarm ausgelöst werden.
- Ein Abschalten des Alarms ist nur mit dem Schlüssel möglich.

Wir wollen zunächst, in gebotener Kürze, auf die wichtigsten Grundlagen der UML (*Unified Modelling Language*) eingehen.

25.1.1 Use Case-Diagramm

Für den Einstieg in die UML-gestützte Programmentwicklung wählen wir das *Use Case Modell*, welches uns hilft, die clientseitigen (d.h. benutzerseitigen) Anforderungen an das zu entwickelnde

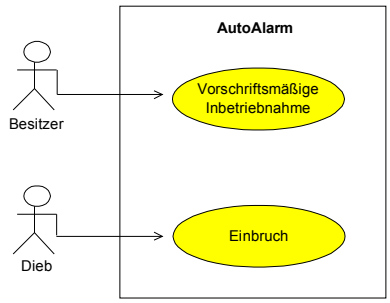
[1] *Unified Modelling Language*

Programm zu bestimmen, um schließlich die an dem zu modellierenden Szenario beteiligten Objekte grob zu identifizieren.

Unter einem Use Case (auch als Anwendungsfall bzw. Geschäftsvorfall bezeichnet) verstehen wir eine typische Handlung, die der Akteur mit dem Programm ausführt. Die einzelnen Use Cases können auch von verschiedenen Akteuren initiiert werden und bestehen aus einzelnen Aktivitäten (Aktionen).

HINWEIS: Das Use Case-Diagramm kann nicht mehr als einen ersten Überblick über die Geschäftsvorfälle geben, da die Abläufe innerhalb eines Prozesses im Einzelnen noch nicht erkennbar sind.

Für unser Beispiel erhalten wir ein sehr einfaches Diagramm:



Um zu obigem Use Case-Diagramm zu kommen, sind folgende Fragen zu beantworten:

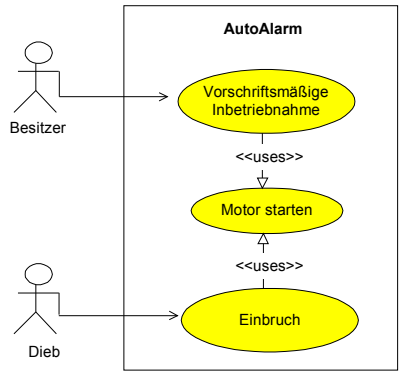
1. Welche Akteure gibt es?
2. Welche typischen Handlungen nehmen sie vor?

Zu 1.: Wir haben es in unserem Modell offensichtlich mit den Akteuren "Besitzer" und "Dieb" zu tun. Hierbei handelt es sich keinesfalls um Personen, sondern um so genannte *Rollen*. Der rechtmäßige Besitzer des Autos kann durchaus die Rolle eines Diebes spielen, nämlich dann, wenn er probeweise versucht, den Motor zu starten, ohne vorher die Alarmanlage entschärft zu haben. Umgekehrt kann ein Dieb dann in die Rolle des Besitzers schlüpfen, wenn er den Fahrzeugschlüssel gestohlen hat.

Zu 2.: Die typischen Handlungen sind "Inbetriebnahme des Autos" (vorgenommen durch den Akteur "Besitzer" und "Versuchter Diebstahl des Autos" (initiiert durch den Akteur "Dieb"). Verlieren Sie sich dabei nicht in einzelne Aktionen, sondern versuchen Sie sich auf die wesentlichen Bedienfunktionen zu konzentrieren.

Falls mehrere Use Cases einen ähnlichen Aufbau haben (d h. ihr Aktionsablauf müsste durch Kopieren wiederholt dargestellt werden) lohnt es sich mitunter, das gemeinsame Verhalten in einen separaten Use Case auszulagern. Man verwendet dazu Include-Beziehungen (<<*uses*>>), die den Zugriff eines Use Case auf einen anderen bezeichnen.

Für unser Beispiel könnte ein Use Case "Motor starten" rein theoretisch in Erwägung gezogen werden.



Allerdings ergäbe in unserem Beispiel der Use Case "Motor starten" wenig Sinn, da es sich hier eher um eine einzelne Aktion handelt (siehe folgender Abschnitt).

HINWEIS: Vermeiden Sie es, Geschäftsvorfälle so weit zu splitten, dass daraus schließlich einzelne Aktionen werden!

25.1.2 Use Case-Dokumentation

Die schriftliche Dokumentation der Use Cases ist erforderlich, um den Ablauf der Aktionen zu spezifizieren:

USE CASE: Inbetriebnahme des Autos

- **Akteur:** Besitzer
- **Überblick:** Zweck ist es, dem Akteur den Start des Motors zu ermöglichen.
- **Bedingung:** Der Akteur verfügt über den Schlüssel.
- **Ablauf der Aktionen**:
  - Akteur richtet den Schlüssel auf das Auto, um dessen Schloss mit seinem Schlüssel zu verbinden.
  - Anschließend betätigt der Akteur die Öffnungstaste des Schlüssels.
  - Autoschloss wechselt vom gesperrten in den geöffneten Zustand, falls die Verbindung zustande gekommen ist.
  - Wenn Autoschloss geöffnet ist, kann Akteur den Motor starten.
  - Akteur schaltet Motor ab.
  - Akteur betätigt Öffnungstaste des Schlüssels, um Autoschloss wieder zu sperren.

USE CASE: Versuchter Diebstahl des Autos

- **Akteur:** Dieb
- **Überblick:** Zweck ist es, dem Akteur den Start des Motors zu verwehren und stattdessen Alarm auszulösen.
- **Bedingung:** Das Auto befindet sich im verschlossenen Zustand.
- **Ablauf der Aktionen**:
  - Der Akteur versucht den Motor zu starten.
  - Es wird Alarm ausgelöst.

25.1.3 Objekte identifizieren

Use Cases identifizieren im Wesentlichen nur die clientseitigen Objekte, also jene, mit denen der Benutzer interagiert.

Hier die wichtigsten Regeln:

- Objekte sind Dinge (gewöhnlich Substantive), manchmal aber können sie auch als Verben ausgedrückt werden.
- Objekte kapseln Daten (gewöhnlich Attribute bzw. Eigenschaften), d h., sie haben einen Zustand.
- Objekte kapseln Verhalten (gewöhnlich Aktionen bzw. Verben), das sind Methoden, die irgendetwas tun.
- Jede Komponente, die bereits identifizierte Objekte enthält, ist ebenfalls ein Objekt.

HINWEIS: Akteure sind nicht Teil des Systems, werden also nicht als Objekte modelliert!

Hilfreich ist es, wenn Sie sich nacheinander alle Use Cases notieren, um die relevanten Ausdrücke gemäß der oben aufgeführten Regeln zunächst einmal zu markieren. Dabei können Sie – ohne Rücksichtnahme auf die Orthografie – die Satzelemente nach Belieben umgruppieren, um die Identifikation als Objekt, Methode, Zustand oder Eigenschaft deutlicher hervortreten zu lassen:

- Besitzer **peilt an** mit **Schlüssel** das **Auto**, um **Schloss** mit **Schlüssel** zu verbinden.
- Besitzer **betätigt die Öffnungstaste** des **Schlüssels**.
- **Schloss wechselt** vom **gesperrten in den geöffneten Schließzustand**, falls die **Verbindung** zustande gekommen ist.
- Wenn **Schloss** geöffnet ist, kann Besitzer den **Motor starten**.
- Besitzer **schaltet ab** den **Motor**.
- Besitzer **drückt Öffnungstaste** des **Schlüssels**, um **Schloss** wieder zu sperren.

- Dieb versucht, den **Motor** zu **starten**.
- Es wird **Alarm** ausgelöst.

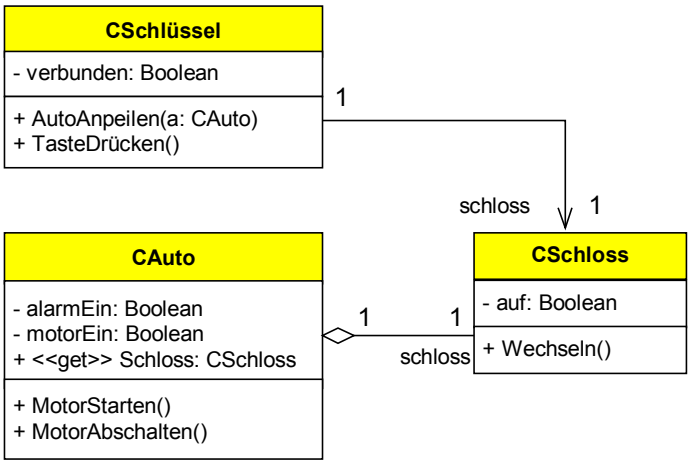
Obwohl obiges Ergebnis jedem Deutschlehrer die Haare zu Berge stehen lässt, kann es in einer nützlichen Tabelle zusammengefasst werden, die wie folgt aussieht:

Objekt	Methoden	Zustände	Objekte/Eigenschaften
Schlüssel	*Auto anpeilen* *Öffnungstaste drücken*	mit Schloss verbunden (ja/nein)	*Schloss*
Schloss	*Schließzustand wechseln*	geöffnet/gesperrt	
Auto			*Schloss* *Motor* *Alarm*
Motor	*Motor starten* *Motor anhalten*	ein/aus	
Alarm	*Alarm auslösen* *Alarm abschalten*	ein/aus	

HINWEIS: Denken Sie immer daran, dass jeder Zustand durch Methodenaufrufe erreichbar sein muss und auch wieder verlassen werden kann!

25.1.4 Statisches Modell

Die folgende Abbildung zeigt die Umsetzung der in den Use Cases identifizierten Objekte in ein Klassendiagramm, das sich aus den Klassen *CAuto*, *CSchlüssel* und *CSchloss* zusammensetzt:



Es handelt sich dabei nur um eine von vielen verschiedenen Möglichkeiten.

HINWEIS: Jedes Objektmodell ist eine persönliche Sicht des Entwicklers auf das Problem!

Wie Sie dem Klassendiagramm entnehmen, wurde auf das Anlegen der Klassen *CMotor* und *CAlarm* verzichtet, stattdessen wurde deren Funktionalität komplett in *CAuto* verlagert. Eine solche Vorgehensweise empfiehlt sich dann, wenn die identifizierten Objekte mit nicht mehr als nur einem anderen Objekt assoziiert sind und eine relativ geringe Komplexität besitzen.

25.1.5 Beziehungen zwischen den Klassen

- Die Klassen *CSchlüssel* und *CSchloss* sind durch eine *binäre Assoziation* miteinander verbunden, die auf beiden Seiten die Kardinalität 1 hat. Der Zugriff ist nur in Pfeilrichtung möglich (der Schlüssel macht etwas mit dem Schloss und nicht umgekehrt!). Damit referenziert (nicht instanziiert!) ein Objekt vom Typ *CSchlüssel* das bereits vorhandene Objekt *schloss* vom Typ *CSchloss*.
- Bei *CAuto* und *CSchloss* handelt es sich um eine typische *Aggregation*, die – wie Sie wissen sollten – eine besondere Form der Assoziation darstellt. Das Objekt *schloss* ist integraler Bestandteil eines Objekts der Aggregat-Klasse *CAuto*, wird also in diesem erzeugt bzw. instanziiert (*new*).

25.1.6 Dynamisches Modell

Als Ergebnis des statischen Modells liegen die identifizierten Klassen mit ihren Zustandsvariablen, Eigenschaften und Methoden vor. Weiterhin wurden dort bereits die Beziehungen zwischen den Klassen (Vererbungsstrukturen, Assoziationen) analysiert. Aufgrund dieser Informationen kann nun das dynamische Verhalten näher spezifiziert werden, wozu sich das Zustandsüberführungsdiagramm (*StateChart*) hervorragend eignet.

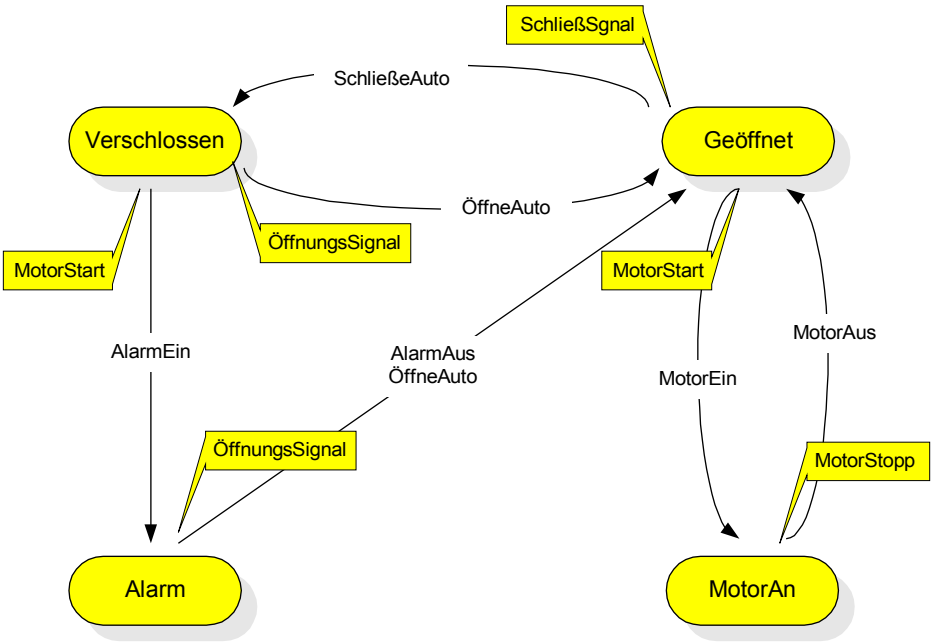
Relevante Zustände identifizieren

Die im statischen Modell enthaltenen Klassen (*CSchlüssel*, *CAuto*, *CSchloss*) besitzen insgesamt drei Zustandsvariablen vom Typ *Boolean* (*auf*, *motorEin*, *alarmEin*). Rein theoretisch ergäben die damit möglichen Kombinationen genau acht verschiedene Zustände. Davon sind allerdings nur vier Zustände von Bedeutung, die übrigen können ausgeschlossen werden (z.B. dürfen *alarmEin* und *motorEin* nicht gleichzeitig *True* sein).

Nr.	Zustand	auf	motorEin	alarmEin
1	Verschlossen	*False*	*False*	*False*
2	Geöffnet	*True*	*False*	*False*
3	MotorAn	*True*	*True*	*False*
4	Alarm	*False*	*False*	*True*

Zustandsüberführungsdiagramm

Ausgangsbasis für die Implementierung des dynamischen Verhaltens ist das Zustandsüberführungsdiagramm, es ist deshalb äußerst sorgfältig aufzustellen. Der Übergang von einem zum anderen Zustand wird immer durch ein Ereignis ausgelöst. Anschließend werden bestimmte Aktionen durchgeführt, hinter denen sich die Methodenaufrufe verbergen.



Im Zustandsüberführungsdiagramm sind enthalten:

- Alle Zustände, die ein *CAuto*-Objekt einnehmen kann.
- Alle Ereignisse, die ein *CAuto*-Objekt empfangen kann (entweder vom Client oder von den über Aggregation eingebundenen Teilobjekten).
- Alle Aktionen, die ein *CAuto*-Objekt als Antwort auf eine Zustand-/Ereignis-Kombination (entsprechend den Pfeilen im Diagramm) ausführen kann. Pro Zustandsübergang können keine, eine oder mehrere Aktionen (im Regelfall sind das Methodenaufrufe in *CAuto*) durchgeführt werden.

25.1.7 Implementierung

Ein fertiges Programm wird nur in seltenen Fällen bereits nach dem ersten Durchlauf vorliegen. Meistens werden Sie statisches und dynamisches Modell iterativ immer wieder verbessern, solange bis ein zufrieden stellendes Ergebnis vorliegt.

HINWEIS: Aus Gründen der Übersichtlichkeit markieren wir die Zustandsvariablen im Quellcode durch einen vorangestellten Unterstrich (_).

Klasse CSchlüssel

```
public class CSchlüssel
{
```

Das folgende Ereignis hat im Objektmodell keine praktische Bedeutung, es dient lediglich zur Auswertung im Testclient, der die Klasse später einmal verwenden soll:

```
    public delegate void SignalEvent(bool v);
    public event SignalEvent Signal;
```

Die Referenz auf das Autoschloss:

```
    private CSchloss _schloss;
```

Auch einem Schlüssel ordnen wir zwei Zustände zu, je nachdem, ob er auf das Schloss gerichtet ist (*verbunden = true*) oder nicht (*verbunden = false*):

```
    private bool _verbunden = false;
```

Die Methode *AutoAnpeilen*(*a: CAuto*) verlangt als Übergabeparameter Typ *CAuto*, denn sie muss "wissen", welches *Auto*-Objekt denn mit dem Schlüssel angepeilt werden soll.

```
    public void AutoAnpeilen(CAuto aut)
    {
        _schloss = aut.Schloss;          // Referenz auf bereits vorhandene Instanz!
        _verbunden = true;
        Signal(_verbunden);              // Ereignis auslösen
    }
```

Die Methode *TasteDrücken()* sorgt nach erfolgreichem Anpeilen für einen Zustandswechsel des *schloss*-Objekts (Aufruf der Methode *Wechseln()*). Um eine realitätsnahe Modellierung zu ermöglichen, machen wir den *verbunden*-Zustand des Schlüssels von einer Zufallszahl abhängig. Mit einer Wahrscheinlichkeit von 1 zu 5 wird die Infrarotverbindung zwischen Schlüssel und Schloss immer wieder mal unterbrochen.

```
    public void tasteDrücken()
    {
        Random rdm = new Random();
        int z = rdm.Next(0, 5);           // erzeugt Zufallszahl aus 0, 1, 2, 3, 4
        if (_verbunden)
        {
            _schloss.Wechseln();                // Schloss wechselt Schließzustand
            if (z == 0) _verbunden = false;     // Verbindung wird zufällig unterbrochen
        }
        else Signal(_verbunden);                // Ereignis auslösen
    }
}
```

Klasse CSchloss

```
public class CSchloss
{
```

Das boolesche Attribut *auf* erlaubt für ein Objekt vom Typ *CSchloss* nur die Zustände "Geöffnet" (*auf = true*) und "Geschlossen" (*auf = false*):

```
    private bool _auf = false;
```

Damit das *CSchloss*-Objekt die Möglichkeit bekommt, das Aggregatobjekt *CAuto* darüber zu informieren, ob das Schloss geöffnet oder geschlossen wurde, werden zwei Ereignisse definiert:

```
    public delegate void SchlossEvent();
    public event SchlossEvent SchlossAuf;
    public event SchlossEvent SchlossZu;
```

Eine Eigenschaft:

```
    public bool Auf
    {
         get {return _auf;}
    }
```

Die Methode *Wechseln()* realisiert den Zustandswechsel des Schlosses von "Geöffnet" nach "Geschlossen" (und umgekehrt) und löst entsprechend die beiden Ereigniss aus:

```
    public void Wechseln()
    {
        if (_auf)
        {
            _auf = false;
            SchlossZu();               // Ereignis auslösen!
        }
        else
        {
            _auf = true;
            SchlossAuf();              // Ereignis auslösen!
        }
    }
}
```

Klasse CAuto

```
public class CAuto
{
```

Der Zustand eines Autos wird beschrieben durch die booleschen Attribute *_alarmEin* und *_motorEin*:

```
    private bool _alarmEin;
    private bool _motorEin;
```

Nur um die Benutzerschnittstelle zu benachrichtigen, werden noch vier Ereignisse zusätzlich benötigt:

```
public delegate void EinAusEvent(bool ein);
public delegate void AlarmEvent();

public event EinAusEvent MotorEinAus;
public event EinAusEvent SchlossWechselt;
public event AlarmEvent AlarmEin;
public event AlarmEvent AlarmAus;
```

Die schreibgeschützte Eigenschaft *Schloss* liefert das komplette Autoschloss (Objekt *schloss*). Dieser Zugriff auf die "Innereien" des Autos ist erforderlich, um das Auto überhaupt öffnen zu können (die Implementierung der Methode *AutoAnpeilen* benötigt eine Referenz auf das Aggregatobjekt *schloss*).

```
private CSchloss _Schloss;
```

Der Lesezugriff auf das Aggregatobjekt:

```
public CSchloss Schloss
{    get {return _Schloss;}   }
```

Der Konstruktor:

```
public CAuto(bool a, bool m)
{
    _Schloss = new CSchloss();            // Teilobjekt (Aggregat) einbinden
    _alarmEin = a; _motorEin = m;
```

Eventhandler anmelden:

```
    _Schloss.SchlossAuf += new CSchloss.SchlossEvent(_Schloss_SchlossAuf);
    _Schloss.SchlossZu += new CSchloss.SchlossEvent(_Schloss_SchlossZu);
}
```

Die Methoden *MotorStarten()* und *MotorAbschalten()* sind in unserem Modell die einzigen öffentlichen Zugriffsmöglichkeiten auf ein *CAuto*-Objekt und bewirken einen Zustandswechsel:

```
public void MotorStarten()
{
    if (_Schloss.Auf)
    {
        _motorEin = true;
        MotorEinAus(_motorEin);   // Ereignis auslösen
    }
    else
    {
        _alarmEin = true;
        AlarmEin();               // Ereignis auslösen
    }
}
```

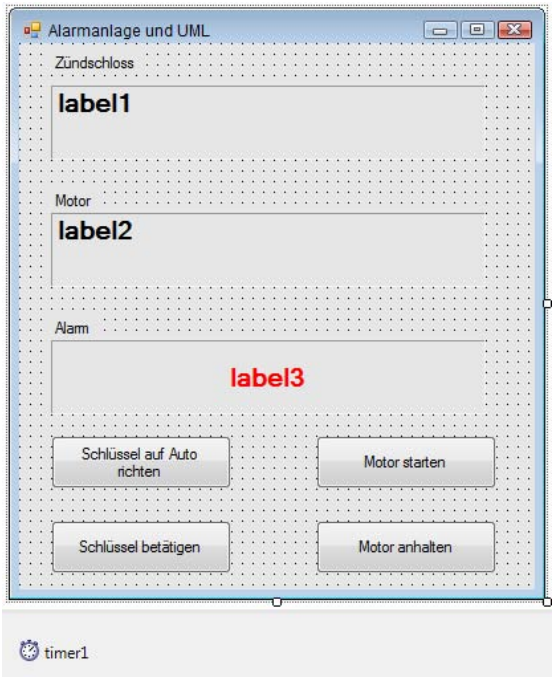
```
public void MotorAbschalten()
{
     _motorEin = false;
    MotorEinAus(_motorEin);          // Ereignis auslösen
}
```

Quasi wie interne Zugriffsmethoden (Callback) wirken die beiden Ereignisbehandlungen *Schloss.SchlossZu* und *Schloss.SchlossAuf*, die durch das *Schloss*-Objekt ausgelöst werden:

```
private void _Schloss_SchlossZu()
{    SchlossWechselt(false); }          // Ereignis auslösen

private void _Schloss_SchlossAuf()
{
    if (_alarmEin)
    {
        _alarmEin = false;
        AlarmAus();                    // Ereignis auslösen
    }
    SchlossWechselt(true);             // Ereignis auslösen
}
```

25.1.8 Test-Client

Um die entwickelten Klassen zu testen, entwerfen wir die abgebildete Bedienoberfläche:



HINWEIS: Die *Timer*-Komponente wird für den Alarm benötigt (blinkendes Label).

```
public partial class Form1 : Form
{
    ...
    private CAuto meinAuto;
    private CSchlüssel meinSchlüssel;
```

Die Aktivitäten beim Laden von *Form1*:

```
    protected override void OnLoad(EventArgs e)
    {
       meinAuto = new CAuto(false, false);
       meinSchlüssel = new CSchlüssel();
       label1.Text = ""; label2.Text = "";  label3.Text = "";
```

Alle Eventhandler anmelden:

```
       meinSchlüssel.Signal += new CSchlüssel.SignalEvent(meinSchlüssel_Signal);
       meinAuto.AlarmEin += new CAuto.AlarmEvent(meinAuto_AlarmEin);
       meinAuto.AlarmAus += new CAuto.AlarmEvent(meinAuto_AlarmAus);
       meinAuto.SchlossWechselt += new CAuto.EinAusEvent(meinAuto_SchlossWechselt);
       meinAuto.MotorEinAus += new CAuto.EinAusEvent(meinAuto_MotorEinAus);
       base.OnLoad(e);
    }
```

Schlüssel auf Auto richten:

```
    private void button1_Click(object sender, EventArgs e)
    {
      meinSchlüssel.AutoAnpeilen(meinAuto);
    }
```

Schlüssel betätigen:

```
    private void button2_Click(object sender, EventArgs e)
    {
      meinSchlüssel.tasteDrücken();
    }
```

Motor starten:

```
    private void button3_Click(object sender, EventArgs e)
    {
       meinAuto.MotorStarten();
    }
```

Motor anhalten:

```
    private void button4_Click(object sender, EventArgs e)
    {
       meinAuto.MotorAbschalten();
    }
```

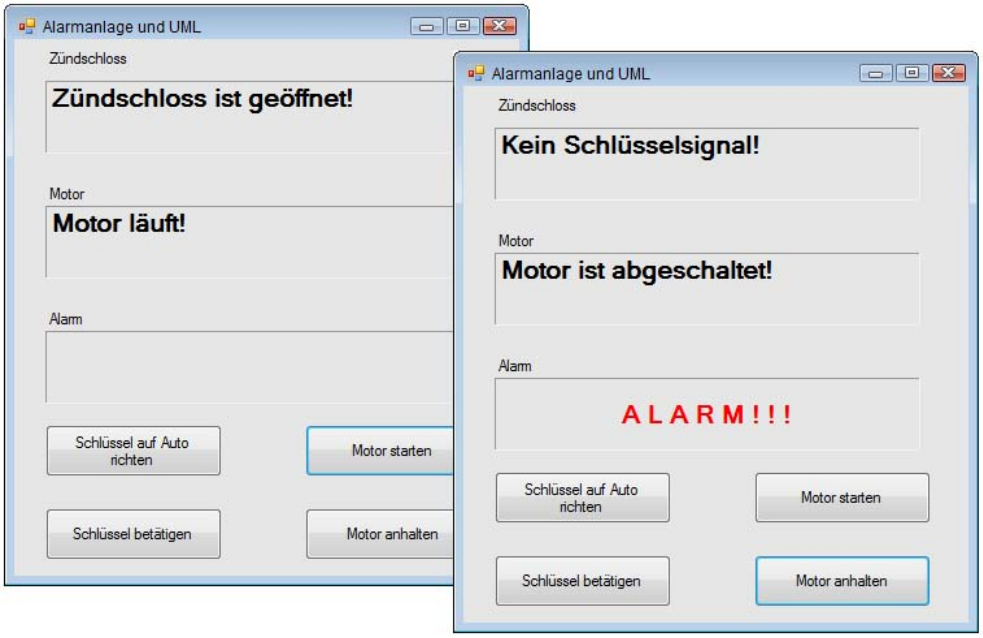
Die diversen Ereignisse auswerten, um sich von der Funktionsfähigkeit des Programms zu überzeugen:

```
    private void meinSchlüssel_Signal(bool v)
    {
         if (v)
              label1.Text = "Zündschloss hat Signal erhalten!";
         else
              label1.Text = "Kein Schlüsselsignal!";
    }

    private void meinAuto_AlarmEin()
    {
         label3.Text = "A L A R M ! ! !"; timer1.Enabled = true;
    }

    private void meinAuto_AlarmAus()
    {
         label3.Text = ""; timer1.Enabled = false;
    }

    private void meinAuto_SchlossWechselt(bool auf)
    {
         if (auf) label1.Text = "Zündschloss ist geöffnet!";
         else  label1.Text = "Zündschloss ist gesperrt!";
    }

    private void meinAuto_MotorEinAus(bool ein)
    {
         if (ein) label2.Text = "Motor läuft!";
         else label2.Text = "Motor ist abgeschaltet!";
    }

    private void timer1_Tick(object sender, EventArgs e)
    {
         label3.Visible = !label3.Visible;
    }
}
```

Test

Jetzt wird es richtig spannend, Sie können wahlweise in die Rolle des rechtmäßigen Besitzers oder in die eines Diebes schlüpfen.

HINWEIS: Vergessen Sie nicht, dass die Schaltfläche "Schlüssel auf Auto richten" aufgrund des eingebauten Zufallsgenerators nicht immer zum Erfolg führt!



Bemerkungen

- Kehren Sie zum Ausgangspunkt des Entwurfs, zum Use Case-Diagramm, zurück und simulieren Sie alle möglichen Benutzereingaben.
- Vergleichen Sie die Programmaktionen mit den verschiedensten Zyklen im Zustandsüberführungsdiagramm!

HINWEIS: Eine alternative Implementierung der Alarmanlage als FSM (*Finite State Machine*) wird im Praxisbeispiel 25.3.1 gezeigt!

25.2 Der Klassen-Designer

Der Klassen-Designer gehört zweifelsfrei mit zu den interessantesten Tools von Visual Studio, er ermöglicht es, die Struktur von Klassen und anderen Typen zu visualisieren und deren Code auf einfache und transparente Weise zu editieren. Das Klassendiagramm kann somit Ihren Code grafisch präsentieren. Änderungen am Klassendiagramm werden sofort im Code sichtbar und umgekehrt.

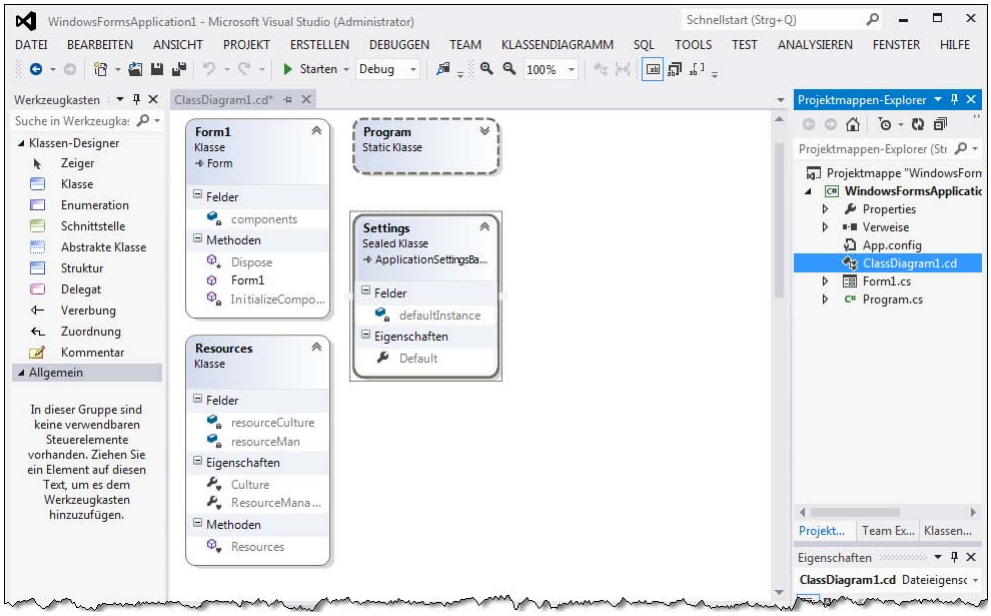
Der Klassen-Designer enthält zahlreiche Features die Ihnen dabei helfen, Ihren Code schneller und besser zu strukturieren. So können z.B. Bezeichner bequem umbenannt werden, Methoden lassen sich einfach überschreiben und Klassen und Strukturen automatisch generieren. Auch Eigenschaften und Schnittstellen lassen sich leicht implementieren, da auch hier die Methodenrümpfe automatisch erstellt werden.

Nicht zu unterschätzen ist die Rolle des Klassen-Designers als Kommunikationstool zwischen den an einem Softwareprojekt beteiligten Programmierern. Die Klassendiagramme können ausgedruckt oder z.B. als HTML Seite oder PowerPoint Präsentation bereitgestellt werden.

HINWEIS: Wer nicht an einer allgemeinen Beschreibung des Klassendesigners interessiert ist, sondern lieber gleich mit der Arbeit beginnen möchte, sollte sich das Praxisbeispiel 25.3.2 näher anschauen!

25.2.1 Ein neues Klassendiagramm erzeugen

Bereits für eine neu erzeugte und ansonsten leere Windows Anwendung können Sie sich sofort das Klassendiagramm betrachten, Sie brauchen dazu nur im Kontextmenü des Projektmappen-Explorers auf *Klassendiagramm anzeigen* zu klicken. Wie es die folgende Abbildung verdeutlicht, zeigt das Klassendiagramm die in jedem neuen Windows Forms Projekt bereits standardmäßig vorhandenen Klassen *Form1*, *Program*, *Ressources* und *Settings*. Links ist der Werkzeugkasten des Klassendesigners zu sehen.



HINWEIS: Beachten Sie auch, dass das Hauptmenü der IDE ein neues Element *Klassendiagramm* bekommen hat.

Da keinerlei Verpflichtung besteht, alle im Projekt vorhandenen Klassen im Klassendiagramm darzustellen, werden Sie in der Regel mit einem leeren Klassendiagramm beginnen und dort nur die für Sie wirklich interessanten Klassen hinzufügen. Auch mehrere Klassendiagramme pro Projekt

sind möglich. Sie selbst treffen die Entscheidung, welche Klassen in welchem Diagramm dargestellt werden sollen!

Um ein neues Klassendiagramm zu Ihrem Projekt hinzuzufügen wählen Sie das Menü *Projekt|Neues Element hinzufügen...* und die Vorlage "Klassendiagramm".

Legen Sie den Namen des Diagramms fest und Visual Studio erzeugt das Element mit dem definierten Namen und einer *.cd* Dateinamenserweiterung.

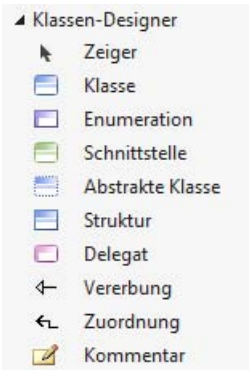
HINWEIS: Das (oder die) Klassendiagramm(e) sind kein Bestandteil Ihres Projekts sondern lediglich Hilfsmittel, d.h., sie werden nicht mit kompiliert!

Sie können vorhandene Klassen auf dem Klassen-Designer darstellen, indem Sie diese einfach vom Projektmappen-Explorer bzw. der Klassenansicht auf die Entwurfsoberfläche ziehen. Außerdem können Sie neue Klassen oder andere Typen hinzufügen, indem Sie diese vom Werkzeugkasten auf den Klassen-Designer absetzen und dort editieren.

HINWEIS: Das Klassendiagramm speichert nur visuelle Informationen und implementiert selbst keinerlei Code. Das Löschen der *.cd*-Datei eines Klassendiagramms führt deshalb zu keinerlei Codeverlusten!

25.2.2 Werkzeugkasten

Wenn der Klassen-Designer geöffnet ist, zeigt der Werkzeugkasten die Elemente, die Sie auf die Oberfläche des Designers ziehen können:



HINWEIS: Im Unterschied zum bekannten Werkzeugkasten für Windows Forms – Anwendungen müssen Sie hier die linke Maustaste gedrückt halten, während Sie ein UML-Element dem Werkzeugkasten auf die Oberfläche des Klassen-Designers ziehen.

Wie Sie sehen, ist die Anzahl der verfügbaren UML-Elemente begrenzt und schöpft bei weitem nicht alle Möglichkeiten der UML aus. Für die Entwicklung von einfachen Klassendiagrammen dürfte das Angebot aber ausreichen.

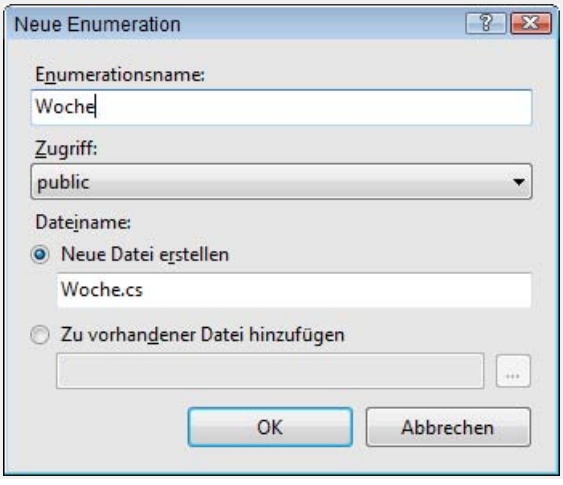
Da man in der UML meist die englischen Originalbezeichner beibehält, zeigt die nachfolgende Tabelle beide Sprachversionen der Werkzeugkasten-Elemente:

eingedeutschte Bezeichnung	englische Originalbezeichnung
Klasse	Class
Enumeration	Enum
Schnittstelle	Interface
Abstrakte Klasse	Abstract Class
Struktur	Struct
Delegat	Delegate
Vererbung	Inheritance
Zuordnung	Association
Kommentar	Comment

25.2.3 Enumeration

Das mit Abstand harmloseste Element, welches Sie mit dem Klassen-Designer erzeugen können, ist die *Enumeration* (eine Sammlung miteinander verwandter Aufzählungstypen). Nachdem Sie dieses Element dem Werkzeugkasten abgezogen und im Designer abgesetzt haben, sind in einem kleinen Dialogfenster zunächst die für alle UML-Elemente typischen Vorentscheidungen zu treffen (Namensvergabe, Zugriffslevel, Dateiname).

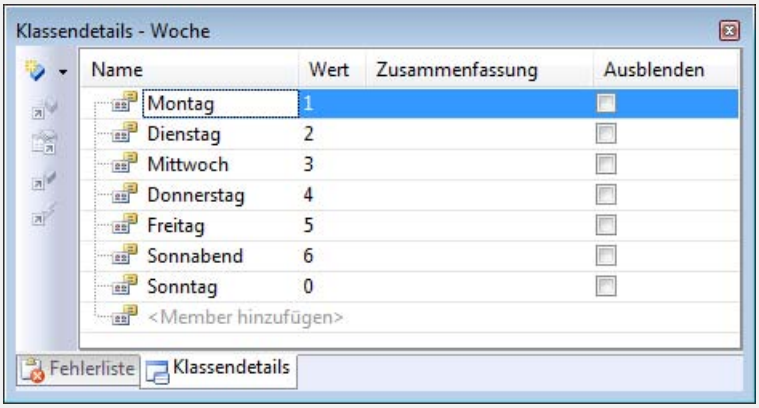
BEISPIEL 25.1: Eine Enumeration für die Tage einer Woche



BEISPIEL 25.1: Eine Enumeration für die Tage einer Woche

Über das Kontextmenü *Hinzufügen|Member* definieren Sie die Mitglieder der Enumeration:

Im Fenster *Klassendetails* (über das Kontextmenü zu öffnen) geben Sie jedem Eintrag einen bestimmten Wert:



Die Datei *Woche.cs* enthält jetzt folgenden Code:

```
public enum Woche
{
    Montag = 1,
    Dienstag = 2,
    Mittwoch = 3,
    Donnerstag = 4,
    Freitag = 5,
    Sonnabend = 6,
    Sonntag = 0,
}
```

BEISPIEL 25.2: Zugriff auf die Enumeration:

C#

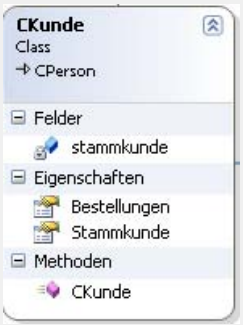
```
const Woche ersterTag = Woche.Montag;
MessageBox.Show(ersterTag.ToString());      // zeigt Montag
```

25.2.4 Klasse

Die Klasse ist zweifelsfrei das wichtigste Element eines Klassendiagramms, sie wird standardmäßig durch ein Rechteck mit vier Abschnitten symbolisiert:

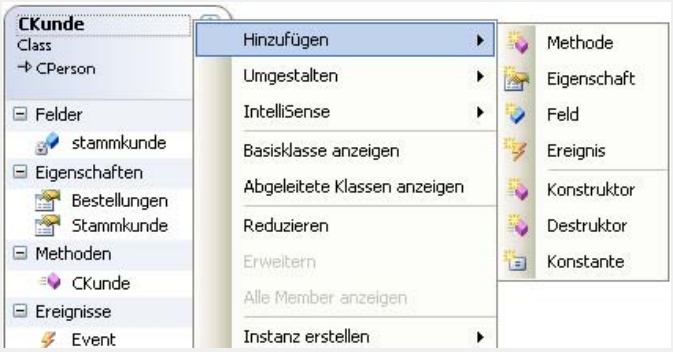
- Name
- Felder
- Eigenschaften
- Methoden

BEISPIEL 25.3: Eine Klasse *CKunde* mit einem Feld, zwei Eigenschaften und einer Methode (Konstruktor).



Um Platz zu sparen, kann die Darstellung der Klasse auf ihren Namen reduziert werden, klicken Sie dazu auf das kleine Symbol in der rechten oberen Ecke:

Über das umfangreiche Kontextmenü einer Klasse lassen sich zahlreiche weitere Optionen einstellen, auch ein Abschnitt für Ereignisse kann hinzugefügt werden.

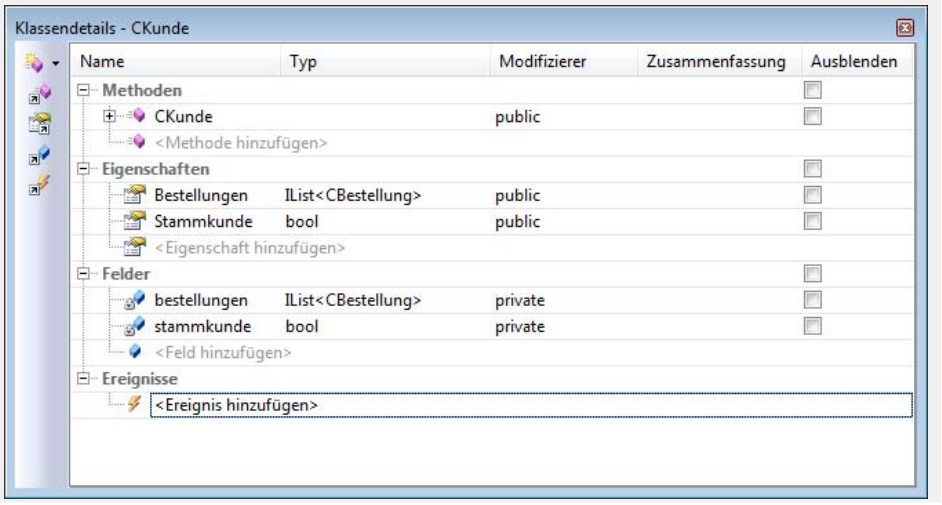


Klassendetails-Fenster

Das Fenster "Klassendetails" öffnen Sie über das Kontextmenü der Klasse (siehe oben). Es handelt sich hierbei um eine editierbare Liste von Feldern, Methoden, Eigenschaften und Ereignissen, wobei Sie wichtige Details eingeben können, wie

- Datentypen,
- Parameter,
- Sichtbarkeitsmodifizierer (*public*, *private*, *internal*, ...),
- Zusammenfassungen bzw. Kommentare zu jedem Klassenmitglied und
- Ausblenden von bestimmten Klassenmitgliedern.

BEISPIEL 25.4: Klassendetails für die Klasse *CKunde* (siehe oben)



Allerdings ist es nicht zwingend erforderlich, dass Sie alle Einträge im Klassendiagramm vornehmen. Die Philosophie des Klassen-Designers basiert darauf, dass Änderungen wahlweise im Code oder im Diagramm vorgenommen werden können. Wenn Sie also z.B. einen bestimmten Datentyp im Code geändert haben, so wird diese Änderung automatisch und sofort auch im Klassendetails-Fenster sichtbar (und umgekehrt)!

HINWEIS: Es ist sowohl von der Problemstellung als auch vom persönlichen Geschmack des Programmierers abhängig, wie er seine Arbeit zwischen Diagramm- und Codefenster aufteilt!

25.2.5 Struktur

*Struktur*en sind komplexe Datentypen, die starke Ähnlichkeiten zu Klassen haben. Im Unterschied zu diesen handelt es sich hier aber nicht um Referenz- sondern um Wertetypen, was die Einsatzszenarien stark beschränkt.

Genauso wie zu einer Klasse können Sie also auch zu einer *Struktur* Felder, Eigenschaften, Methoden, Ereignisse und Konstruktoren hinzufügen.

BEISPIEL 25.5: Eine Struktur *Kunde*

C#

Der automatisch generierte Code:

```
public struct Kunde
{
        private int stammkunde;

        public int Stammkunde
        {
            get { return stammkunde; }
            set { stammkunde = value; }
        }

        public Kunde()
        {
            throw new System.NotImplementedException();
        }
}
```

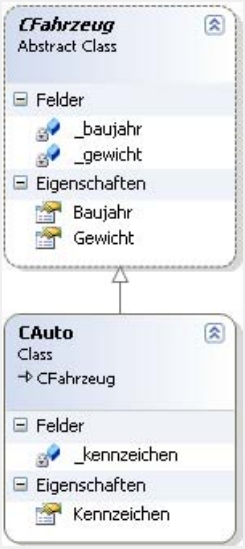
HINWEIS: Da die Implementierung des Konstruktors noch offen ist, wird eine *System.NotImplementedException* ausgelöst.

25.2.6 Abstrakte Klasse

Von einer abstrakten Klasse dürfen keine Instanzen gebildet werden, sie ist lediglich dazu da, ihr "Erbmaterial" an die Nachkommen weiterzugeben. Rein optisch ist sie im Designer u.a. dadurch zu erkennen, dass der Klassenname kursiv gedruckt ist.

BEISPIEL 25.6: Eine abstrakte Klasse *CFahrzeug* mit dem Nachkommen *CAuto*

C#



Der vom Designer generierte Code:

```
public abstract class CFahrzeug
{
        private int _baujahr;

        public int Baujahr
        {
            get { return _baujahr; }
            set { _baujahr = value; }
        }
        private int _gewicht;

        public int Gewicht
        {
            get { return _gewicht; }
            set { _gewicht = value; }
        }
}
```

Die abgeleitete Klasse *CAuto*:

```
public class CAuto : CFahrzeug
{
        private int _kennzeichen;

        public int Kennzeichen
        {
            get { return _kennzeichen; }
```

BEISPIEL 25.6: Eine abstrakte Klasse *CFahrzeug* mit dem Nachkommen *CAuto*

```
            set { _kennzeichen = value; }
        }
}
```

25.2.7 Schnittstelle

Eine Schnittstelle (Interface) können Sie sich wie eine "leere" abstrakte Klasse vorstellen, die lediglich Deklarationen (also keine Implementierungen) von Methoden, Eigenschaften und Ereignissen enthält. Jede andere Klasse, die von dieser Schnittstelle "erbt", übernimmt auch die Verpflichtung, ausnahmslos alle Schnittstellenmitglieder zu implementieren. Weitere eigene Mitglieder können hinzugefügt werden.

HINWEIS: Eine Klasse kann zwar von mehreren Schnittstellen, aber nur von einer einzigen Klasse erben[1]!

BEISPIEL 25.7: Entwurf einer Schnittstelle *IPerson* mit drei Eigenschaften und einer Methode

C#

Der Code der automatisch angelegten Schnittstelle *IPerson.cs:*

```
public interface IPerson
{
        string Nachname
        {
            get;
            set;
        }

        string Anrede
        {
            get;
            set;
        }
        string Wohnort
        {
```

[1] Mehrfachvererbung ist, und das aus gutem Grund, unter .NET nach wie vor tabu.

BEISPIEL 25.7: Entwurf einer Schnittstelle *IPerson* mit drei Eigenschaften und einer Methode

```
            get;
            set;
        }
        string getAdresse();
}
```

Fügen Sie eine Klasse *CStudent* hinzu und zeichnen Sie eine Vererbungsbeziehung von *CStudent* nach *CPerson*. Im Ergebnis übernimmt die Klasse *CStudent* alle Schnittstellenelemente von *IPerson*:

Wenn wir jetzt einen Blick auf den Quellcode von *CStudent* werfen, sehen wir, dass alle Eigenschaften- und Methodenrümpfe vorbereitet sind. Bevor wir aber mit dem Implementieren beginnen können, wollen wir noch entsprechende Felder (*_Anrede, _Nachname, _Wohnort*) hinzufügen, auf welche die einzelnen Methoden zugreifen können.

Um den Unterschied zwischen den von der Schnittstelle "geerbten" und "eigenen" Klassenmitgliedern zu verdeutlichen, fügen wir noch ein privates Feld *_Semester* hinzu, dessen Umwandlung in eine Eigenschaft wir über das Kontextmenü *Umgestalten|Feld einkapseln...* automatisch vornehmen lassen können:

Ein abschließender Blick auf das "Gemeinschaftswerk" von Klassen-Designer und Entwickler:

```
public class CStudent : IPerson
{
        private int _Anrede;
        private int _Nachname;
        private int _Wohnort;
        private int _Semester;

        public int Semester
        {
            get { return _Semester; }
            set { _Semester = value; }
        }
```

Alle Schnittstellenmethoden hat der Klassen-Designer in eine Region *IPerson Member* eingeschlossen:

```
        #region IPerson Member
        public string Nachname
        {
            get{ return _Nachname; }
            set{ _Nachname = value; }
        }

        public string Anrede
        {
            get { return _Anrede; }
            set {_Anrede = value; }
        }

        public string Wohnort
        {
            get {return _Wohnort; }
            set {_Wohnort = value; }
        }

        public string getAdresse()
        {
            return (_Anrede + "  " + _Nachname + "  " + _Wohnort);
        }
        #endregion
}
```

25.2.8 Delegate

Ein Delegate charakterisiert den Typ einer Methode (ähnlich wie eine Klasse den Typ eines Objekts darstellt).

BEISPIEL 25.8: Delegate für Methoden, die zwei Integer-Zahlen als Parameter entgegennehmen und das Ergebnis als Integer-Zahl zurückgeben, soll bereitgestellt und angewendet werden.

C#

Nachdem wir das Delegate-Element auf die Oberfläche des Klassen-Designers gezogen haben, legen wir im Eröffnungsdialog fest, dass der Delegate keine eigene Datei erhält, sondern mit in die Datei *Form1.cs* aufgenommen wird.

Per Kontextmenü *Hinzufügen|Parameter* werden die Parameter *x* und *y* eingetragen:

Im Fenster *Klassendetails* erfolgt das Festlegen des Delegaten und der Parametertypen (*int*):

Name	Typ	Modifizierer	Zusammenfassung
◢ calcDlg	int	public	
(x	int	Keine	
, y	int	Keine	
) <Parameter hinzufügen>			

Der Designer hat den Delegaten in *Form1.cs* eingetragen:

```
namespace Test
{
    public delegate int calcDlg(int x, int y);     // vom Designer

    public partial class Form1 : Form
    {
       ...
```

Die beiden Methoden zum Addieren und Multplizieren von Integer-Zahlen:

```
      private int addInt(int a, int b)
       { return (a + b); }

       private int multInt(int a, int b)
       {  return (a * b); }
```

Die Signatur der Methoden *addInt* und *multInt* muss der Signatur (dem Typ) des Delegaten entsprechen!

```
       private void button1_Click(object sender, EventArgs e)
       {
```

Eine Methodenreferenz auf den Delegaten:

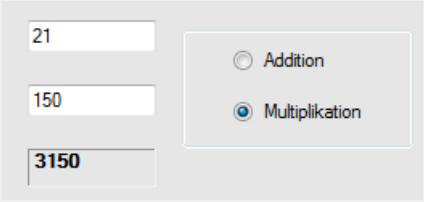
```
          calcDlg calcMethod;
```

BEISPIEL 25.8: Delegate für Methoden, die zwei Integer-Zahlen als Parameter entgegennehmen und das Ergebnis als Integer-Zahl zurückgeben, soll bereitgestellt und angewendet werden.

In Abhängigkeit von der gewünschten Operation (Addition/Multiplikation) wird der Delegate mit der Methode *addInt* oder *multInt* instanziiert[1]:

```
            if (radioButton1.Checked) calcMethod = addInt; else  calcMethod = multInt;

            int z1 = Convert.ToInt32(textBox1.Text);
            int z2 = Convert.ToInt32(textBox2.Text);
            label1.Text = calcMethod(z1, z2).ToString();
        }
    }
}
```

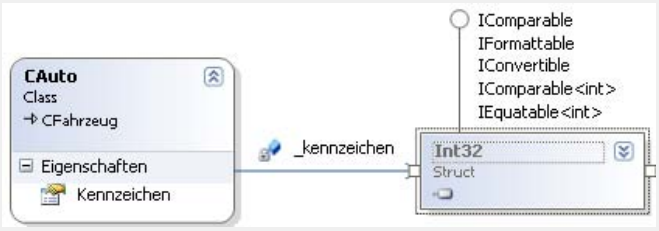
Ergebnis



25.2.9 Zuordnung

Felder kann man entweder als *Feld* oder als *Zuordnung* (Assoziation) darstellen (Kontextmenü *Als Zuordnung anzeigen* bzw. *Als Feld anzeigen*).

BEISPIEL 25.9: Das Feld *_kennzeichen* der Klasse *CAuto* wird als *Zuordnung* zu seinem Datentyp angezeigt (es erscheint demnach nicht mehr im Klassendiagramm).



Es gibt auch eine *Auflistungszuordnung* (*Collection Association*) für Listen (Kontextmenü *Als Auflistungszuordnung anzeigen*), diese funktioniert allerdings nur bei parametrisierten Listen (Generics).

HINWEIS: Ein ausführliche Demonstration des Klassen-Designers mit Vererbung und Auflistungszuordnungen finden Sie im Praxisbeispiel 25.3.2!

[1] Die Instanziierung eines Delegaten kann auch ohne *new*-Operator erfolgen!

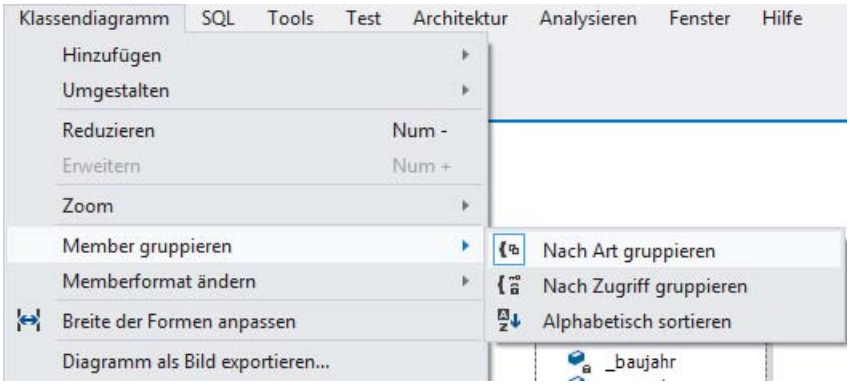
25.2.10 Vererbung

Um die Basisklasse einer Unterklasse anzuzeigen, klicken Sie mit der rechten Maustaste in den Kopfbereich der Klasse und wählen im Kontextmenü *Basisklasse anzeigen*.

Es gibt auch einen umgekehrten Weg: Um die Klassen anzuzeigen, die von einer existierenden Klasse erben, wählen Sie im Kontextmenü *Abgeleitete Klassen anzeigen*. Die erbenden Klassen erscheinen im Diagramm, verbunden mit einer Vererbungslinie (siehe obige Beispiele).

25.2.11 Diagramme anpassen

Die Anzeigeeigenschaften der Klassendiagramme sind einstellbar, die meisten davon über das Hauptmenü *Klassendiagramm*.



Die folgende Abbildung zeigt die Auswirkungen des Menübefehls *Member gruppieren*:

- Nach Art gruppieren (links)
- Nach Zugriff gruppieren (Mitte)
- Alphabetisch sortieren (rechts)

25.2.12 Wann lohnt sich der Klassen-Designer?

Der Klassen-Designer erleichtert Ihnen den Einstieg in die UML, allerdings ist dies nur ein Anfang, da die UML viel mehr leisten kann. Als reines Entwurfshilfsmittel stellt der Klassen-Designer zusätzlich zum Codefenster eine weitere Sicht auf das Projekt zur Verfügung. Änderungen im Code und im Klassendiagramm sind immer synchron!

Eine gleichwertige Alternative zum Codefenster kann der Klassen-Designer natürlich nicht sein, da er lediglich Typen, Methodenrümpfe und -signaturen und deren Beziehungen visualisiert. Die konkrete Implementierung bleibt nach wie vor Sache des Codefensters!

Pro Projekt sind mehrere Klassendiagramme möglich, wobei die folgende Empfehlung gilt:

HINWEIS: Verwenden Sie lieber viele einfache als nur wenig komplizierte Klassendiagramme!

Selbst für bislang hartnäckige OOP-Verweigerer dürfte der Einsatz des Klassen-Designers ein verlockendes Angebot sein, sich die Technologie der UML-Klassendiagramme ohne viel Aufwand anzueignen. So kann man z.B. schnell aufschlussreiche Erkenntnisse gewinnen, wenn man sich zu seinen bereits vorhandenen Projekten nachträglich Klassendiagramme generieren lässt und diese analysiert.

25.3 Praxisbeispiele

25.3.1 Implementierung einer Finite State Machine

Die im Abschnitt 25.1 dieses Kapitels erörterte Implementierung einer "Auto-Alarmanlage" zeigt die Schwierigkeiten, bereits bei relativ wenig Zuständen und Ereignissen die Übersicht zu behalten und Programmierfehler vermeiden. Die folgende systematische Vorgehensweise übergibt die Steuerung des dynamischen Verhaltens an eine FSM (*Finite State Machine* = Endlicher Zustandsautomat).

HINWEIS: Mit der FSM steht uns ein universelles Pattern für die Implementierung nahezu beliebiger objekt- und ereignisgesteuerter Vorgänge zur Verfügung!

Zustandsüberführungsmatrix

Der erste Schritt ist in der Regel die Aufstellung eines Zustandsüberführungsdiagramms, wie es bereits ausführlich in Abschnitt 25.1.6 ausführlich diskutiert wurde.

Der zweite Schritt ist das Umsetzen dieses Diagramms in eine Matrix, wobei die Zeilen die einzelnen Zustände und die Spalten die einzelnen Ereignisse sind. In jeder Zelle sind die auszuführenden Aktionen (kursiv) und der Folgezustand (fett) angegeben.

	ÖffnungsSignal	SchließSignal	MotorStart	MotorStopp
Verschlossen	*ÖffneAuto*		*AlarmEin*	
	Geöffnet		**Alarm**	
Geöffnet		*SchließeAuto*	*MotorEin*	
		Verschlossen	**MotorAn**	
MotorAn				*MotorAus*
				Geöffnet
Alarm	*AlarmAus* *ÖffneAuto*			
	Geöffnet			

Da unser Zustandsdiagramm sechs Pfeile (Zustandsüberführungen bzw. Transitionen) enthält, sind in der Tabelle sechs Zellen besetzt. Die zehn freien Felder bedeuten, dass diese Zustands-/Ereigniskombinationen in unserem Modell nicht vorkommen.

Die FSM ist nichts weiter als die programmtechnische Umsetzung obiger Zustandsüberführungstabelle. Wir realisieren sie als abstrakte Klasse, da von ihr keine Instanz gebildet wird (sie vererbt lediglich an *CAuto*).

```
public abstract class CAutoFSM        // vererbt an CAuto
{
```

Zunächst legen wir die Datenstruktur der FSM fest und orientieren uns dabei ausschließlich an obiger Tabelle. Alle Zustände:

```
    protected enum autoZ
    {
        Verschlossen,      // initialer Zustand    (0)
        Geöffnet,
        MotorAn,
        Alarm,
        MAX                // max. Anzahl          (4)
    }
```

Alle Ereignisse:

```
    protected enum autoE
    {
        SchliessSignal,        //                (0)
        ÖffnungsSignal,
        MotorStart,
        MotorStopp,
        MAX                    // max. Anzahl     (4)
    }
```

Alle Aktionen:

```
protected enum autoA
{
     SchliesseAuto,       //  (0)
     ÖffneAuto,
     AlarmEin,
     AlarmAus,
     MotorEin,
     MotorAus            //   (5)
}
```

Pro Zustandsübergang können keine, eine oder mehrere Aktionen durchgeführt werden. Jeder Übergang entspricht einem Pfeil im Diagramm, gilt also immer für eine ganz bestimmte Zustand-/Ereigniskombination.

```
protected struct aktionsListe
{
```

Die auszuführenden Aktionen werden in einem Array veränderlicher Größe gespeichert:

```
    public autoA[] aktion;
```

Der Folgezustand:

```
    public autoZ nextState;
}
```

Das Implementieren der FSM beginnt mit dem Deklarieren des zweidimensionalen Arrays für die Zustandsüberführungsmatrix:

```
private aktionsListe[,] ZTab = new aktionsListe[(int) autoZ.MAX, (int) autoE.MAX];
private CAuto _auto;
```

Die Referenz auf das *CAuto*-Objekt (erforderlich, da die Aktionen dort aufgerufen werden):

```
public void setAuto(CAuto a)
{
     _auto = a;
}
```

Die Zustandsüberführungsmatrix wird im Konstruktor der Klasse implementiert. Für jede Zustand-/Ereignis-Kombination werden die durchzuführenden Aktionen und der Folgezustand (entsprechend obiger Tabelle) hart codiert, was eine rein routinemäßige Arbeit ist:

```
public CAutoFSM()
{
```

Wir beginnen mit dem Zustand "Verschlossen", auf den laut obiger Tabelle die Ereignisse "Öffnungssignal" und "MotorStart" treffen können:

```
    int z = (int) autoZ.Verschlossen;
    int e = (int) autoE.ÖffnungsSignal;
    ZTab[z, e].aktion = new autoA[1] {autoA.ÖffneAuto};          // Aktion
    ZTab[z, e].nextState = autoZ.Geöffnet;                       // Folgezustand
```

```
        e = (int) autoE.MotorStart;
        ZTab[z, e].aktion = new autoA[1] {autoA.AlarmEin};          // Aktion
        ZTab[z, e].nextState = autoZ.Alarm;                         // Folgezustand
```

Der Zustand "Geöffnet":

```
        z = (int) autoZ.Geöffnet;
        e = (int) autoE.SchliessSignal;
        ZTab[z, e].aktion = new autoA[1] {autoA.SchliesseAuto};
        ZTab[z, e].nextState = autoZ.Verschlossen;
        e = (int) autoE.MotorStart;
        ZTab[z, e].aktion = new autoA[1] {autoA.MotorEin};
        ZTab[z, e].nextState = autoZ.MotorAn;
```

Der Zustand "MotorAn":

```
        z = (int) autoZ.MotorAn;
        e = (int) autoE.MotorStopp;
        ZTab[z, e].aktion = new autoA[1] {autoA.MotorAus};
        ZTab[z, e].nextState = autoZ.Geöffnet;
```

Der Zustand "Alarm":

```
        z = (int) autoZ.Alarm;
        e = (int) autoE.ÖffnungsSignal;
        ZTab[z, e].aktion = new autoA[2] {autoA.AlarmAus, autoA.ÖffneAuto};
        // zwei Aktionen!
        ZTab[z, e].nextState = autoZ.Geöffnet;
    }
```

Die Zustandsüberführungsfunktion übernimmt als Parameter den aktuellen Zustand und das empfangene Ereignis und steuert den Aufruf der Aktionen. Rückgabewert ist immer der Folgezustand:

```
    protected autoZ Ausführen(autoZ az, autoE ae)
    {
        int z = (int) az, e = (int) ae;
        try
        {
```

Nacheinander alle Aktionen für die übergebene Zustand-/Ereignis-Kombination aufrufen, jede Aktion entspricht einem Methodenaufruf in *CAuto*:

```
            for (int i = 0; i < ZTab[z, e].aktion.Length; i++)
            {
                switch (ZTab[z, e].aktion[i])
                {
                    case autoA.ÖffneAuto: _auto.autoÖffnen(); break;
                    case autoA.SchliesseAuto: _auto.autoSchließen(); break;
                    case autoA.MotorEin: _auto.motorEinschalten(); break;
                    case autoA.MotorAus: _auto.motorAusschalten(); break;
```

```
                    case autoA.AlarmEin: _auto.alarmEinschalten(); break;
                    case autoA.AlarmAus: _auto.alarmAusschalten(); break;
                }
            }
```

Den Folgezustand aus der Tabelle entnehmen und zurückgeben:

```
            return ZTab[z, e].nextState;
        }
```

Falls die Zustand-/Ereigniskombination nicht in der Tabelle enthalten ist, wird der Zustand unverändert zurückgegeben:

```
        catch
        {  return az;  }
    }
}
```

CAuto

Die Implementierung von *CAuto* ist übersichtlich, da das gesamte Zustandsmanagement von der FSM bereitgestellt wird:

```
public class CAuto : CAutoFSM               // Vererbung!
{
    private CSchloss _Schloss; // Referenz auf Aggregatobjekt
```

Die einzige Zustandsvariable wird von der FSM verwaltet:

```
    private autoZ _zustand;
```

Der Konstruktor (wird nach dem Konstruktor von *CAutoFSM* aufgerufen):

```
    public CAuto()
    {
        _Schloss = new CSchloss();          // Teilobjekt (Aggregat) einbinden
        setAuto(this);                      // Auto-Objekt an FSM übergeben
        _zustand = autoZ.Verschlossen;      // initialer Zustand
         // Eventhandler anmelden:
        _Schloss.SchlossAuf += new CSchloss.SchlossEvent(_Schloss_SchlossAuf);
        _Schloss.SchlossZu += new CSchloss.SchlossEvent(_Schloss_SchlossZu);
    }
```

Lesezugriff auf das Aggregat:

```
    public CSchloss Schloss
    {
       get {return _Schloss;}
    }
```

Senden von Ereignissen an FSM(Input), öffentliche Methoden (durch Client-Ereignisse aufgerufen):

```
    public void MotorStarten()
    {  _zustand = Ausführen(_zustand, autoE.MotorStart);    }

    public void MotorAbschalten()
    {  _zustand = Ausführen(_zustand, autoE.MotorStopp);   }
```

Private Methoden (durch *CSchloss*-Ereignisse aufgerufen):

```
    private void _Schloss_SchlossZu()
    {  _zustand = Ausführen(_zustand, autoE.SchliessSignal); }

    private void _Schloss_SchlossAuf()
    { _zustand = Ausführen(_zustand, autoE.ÖffnungsSignal); }
```

Die folgenden Ereignisse dienen lediglich der Benachrichtigung des Clients und haben im Objektmodell keinerlei Bedeutung:

```
    public delegate void EinAusEvent(bool ein);
    public delegate void AlarmEvent();

    public event EinAusEvent SchlossWechselt;
    public event EinAusEvent MotorEinAus;
    public event AlarmEvent AlarmEin;
    public event AlarmEvent AlarmAus;

    internal void autoÖffnen()
    {  SchlossWechselt(true);   }

    internal void autoSchließen()
    { SchlossWechselt(false);    }

    internal void motorEinschalten()
    { MotorEinAus(true);  }

    internal void motorAusschalten()
    { MotorEinAus(false);  }
    internal void alarmEinschalten()
    { AlarmEin();  }

    internal void alarmAusschalten()
    { AlarmAus();   }
}
```

Clientcode

Hier sind dank OOP null Änderungen gegenüber der Implementierung nach Abschnitt 25.1 erforderlich, denn die Client-Objekte brauchen "keine Ahnung" davon zu haben, **wie** Sie den Zustand des Server-Objekts implementiert haben, Hauptsache die Schnittstelle bleibt unverändert!

Test

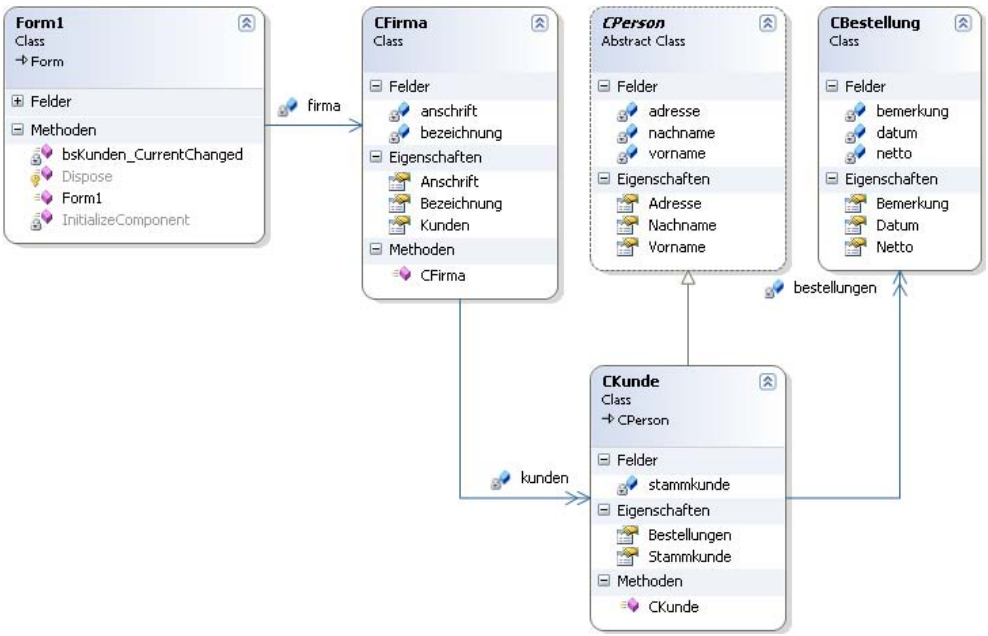
Der Aufwand hat sich gelohnt – unsere FSM funktioniert! Sie werden rein äußerlich keinerlei Unterschied zur klassischen Implementierung des Objektmodells nach 25.1 feststellen.

25.3.2 Modellierung des Bestellsystems einer Firma

Der Klassen-Designer gehört mit zu den bemerkenswertesten Features von Visual Studio. Viele Entwickler, die bisher vor dem Einsatz von UML-Diagrammen zurückschreckten, können sich mit diesem Tool den Einstieg in den objekt- und schichtenorientierten Entwurf deutlich erleichtern. Das folgende Beispiel demonstriert nicht nur die Anwendung des Klassen-Designers, auch andere Features, wie generische Listen, *BindingSource*, *BindingNavigator* und *DataGridView*, kommen zum Einsatz.

Klassendiagramm

Damit gleich zu Beginn klar wird, in welche Richtung wir uns bewegen wollen, soll das anzustrebende Ergebnis bereits jetzt gezeigt werden.



Das abgebildete Klassendiagramm modelliert ein Projekt, welches die Kunden einer Firma und die von jedem Kunden aufgegebenen Bestellungen verwaltet. Dabei werden grundlegende UML-Elemente eingesetzt, wie Vererbung und Assoziationen mit einfacher und mehrfacher Kardinalität (Multiplizität).

Die Benutzerschnittstelle *Form1* ist über die Assoziation *firma* mit der Geschäftslogik verbunden. Diese besteht aus den Klassen *CFirma*, *CKunde* und *CBestellung* sowie der abstrakten Klasse

CPerson, welche an *CKunde* vererbt. Die Assoziationen zwischen den Klassen *CFirma* und *CKunde*, sowie zwischen *CKunde* und *CBestellung*, fallen durch doppelte Pfeilspitzen auf. Hier handelt es sich um Assoziationen der Multiplizität 0..*, die in der deutschen Übersetzung etwas verwirrend als "Auflistungszuordnungen" bezeichnet werden. In unserem Beispiel bedeutet das, dass eine Firma keinen, einen oder viele Kunden und ein Kunde keine, eine oder viele Bestellungen haben kann.

Um obiges Klassendiagramm in ein funktionierendes Programm umzusetzen öffnen Sie eine neue Windows Anwendung mit dem Namen *Firma1*. Das Startformular *Form1* bleibt zunächst unbeachtet liegen, denn die Benutzerschnittstelle liefert nur das i-Tüpfelchen unserer Anwendung und kommt deshalb erst ganz zum Schluss an die Reihe.

Wählen Sie das Menü *Projekt|Neues Element hinzufügen ...* und klicken Sie im Dialogfenster auf die Vorlage *Klassendiagramm*. Der standardmäßige Namen *ClassDiagram1.cd* kann beibehalten werden.

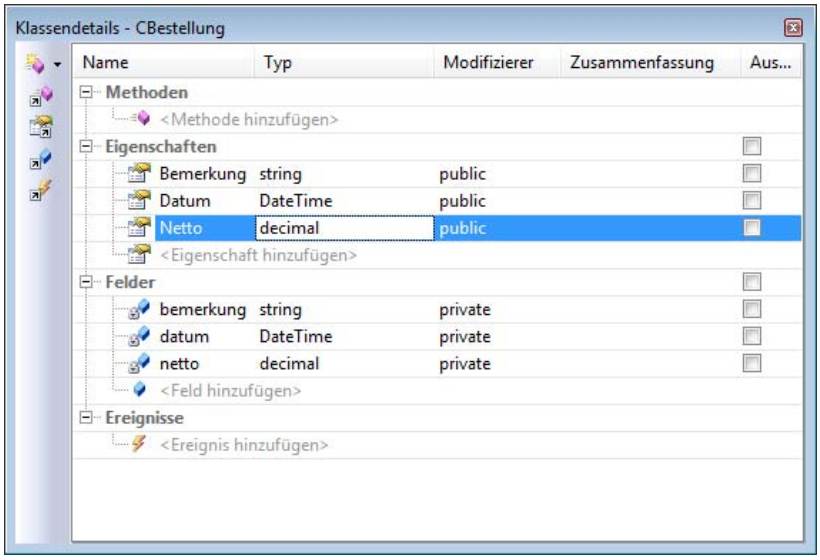
Klasse CBestellung

Beginnen Sie zweckmäßigerweise mit dem relativ einfachen Diagramm der Klasse *CBestellung*. Diese Klasse repräsentiert eine einzelne Bestellung und hat die drei Felder *datum*, *netto* und *bemerkung*, welche durch gleichlautende Eigenschaften gekapselt sind.

Ziehen Sie von der Toolbox eine "Klasse" auf die Oberfläche des Klassen-Designers, vergeben Sie im Dialogfeld den Namen *CBestellung* und legen Sie den Zugriff auf *public* fest.

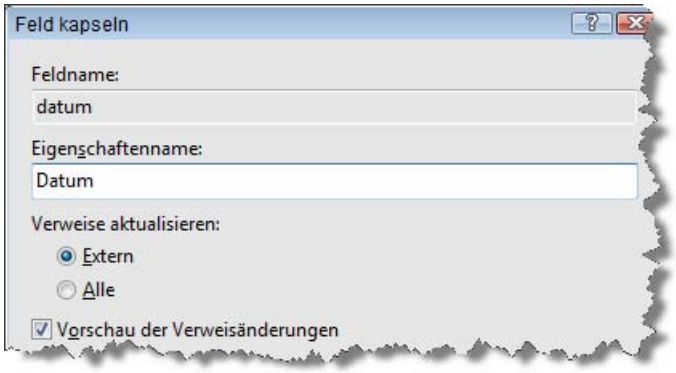
Über das Kontextmenü der rechten Maustaste werden die Felder *datum*, *netto* und *bemerkung* zur Klasse hinzugefügt.

Vergessen Sie nicht, jedem Feld seinen Datentyp zuzuweisen (standardmäßig *int*). Klicken Sie dazu mit der rechten Maustaste auf das Feld und öffnen Sie dann per Kontextmenü das Fenster *Klassendetails*. In die "Typ"-Spalte tragen Sie den passenden Datentyp ein, wobei die Intellisense bei der Auswahl hilft:



Schließlich müssen die drei Felder noch als Eigenschaften gekapselt werden. Klicken Sie auf das entsprechende Feld und wählen Sie das Kontextmenü *Umgestalten|Feld einkapseln...:*

Im anschließenden Dialog können Sie dann den endgültigen Namen der Eigenschaft festlegen:



Nachdem die erste Klasse fertig gestellt ist, sei uns ein neugieriger Blick auf den vom Klassen-Designer automatisch generierten Code erlaubt (Kontextmenü *Code anzeigen* der Klasse im Designer oder im Projektmappen-Explorer):

```
namespace Firma1
{
    public class CBestellung
    {
        private decimal netto;
        public decimal Netto
        {
            get { return netto; }
            set { netto = value; }
        }

        private DateTime datum;
        public DateTime Datum
        {
            get { return datum; }
            set { datum = value; }
        }

        private string bemerkung;
        public string Bemerkung
        {
            get { return bemerkung; }
            set { bemerkung = value; }
        }
    }
}
```

Klasse CPerson

Diese Klasse wird nicht instanziiert, sondern sie soll lediglich ihr "Erbmaterial" an untergeordnete Klassen (in unserem Fall *CKunde*) weitergeben. Von der Toolbox ziehen Sie das Element "Abstrakte Klasse" auf die Oberfläche des Designers und vergeben dafür den Namen *CPerson*. Fügen Sie die *string*-Felder *vorname*, *nachname*, *adresse* hinzu und kapseln Sie diese als Eigenschaften, die Vorgehensweise ist dieselbe wie bei der Klasse *CBestellungen*.

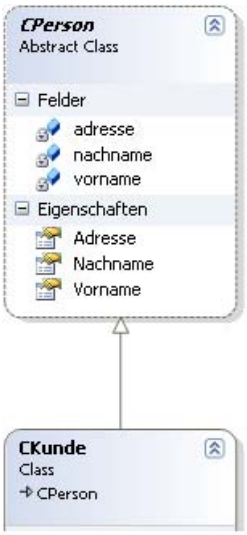
Der automatisch generierte Code der Klasse:

```
public abstract class CPerson
{
        private string vorname;
        public string Vorname
        {
            get { return vorname; }
            set { vorname = value; }
        }
```

```
        private string nachname;
        public string Nachname
        {
            get { return nachname; }
            set { nachname = value; }
        }

        private string adresse;
        public string Adresse
        {
            get { return adresse; }
            set { adresse = value; }
        }
}
```

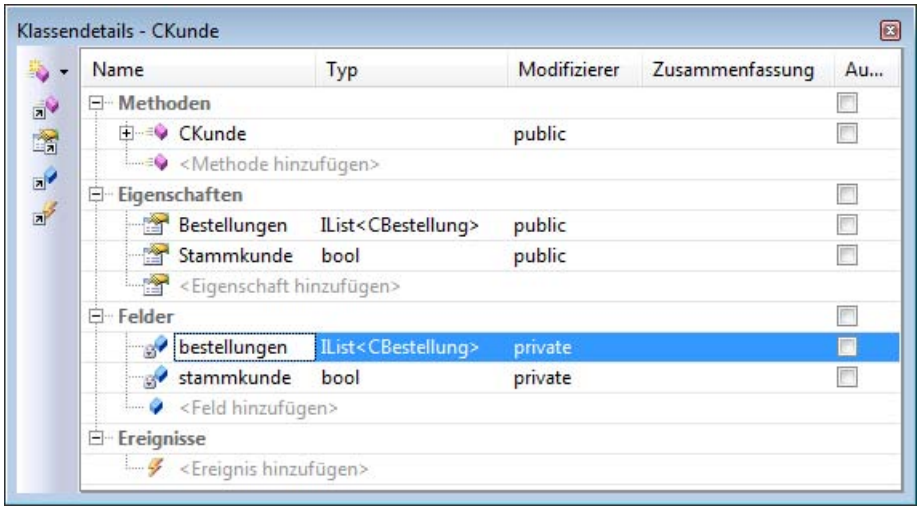
Klasse CKunde

Unterhalb von *CPerson* platzieren Sie eine normale "Klasse" mit dem Namen *CKunde*. Wählen Sie in der Toolbox das Element "Vererbung" und ziehen Sie damit eine Linie von *CKunde* nach *CPerson*. Beide Klassen sind dann durch eine Vererbungsbeziehung wie folgt miteinander verbunden:

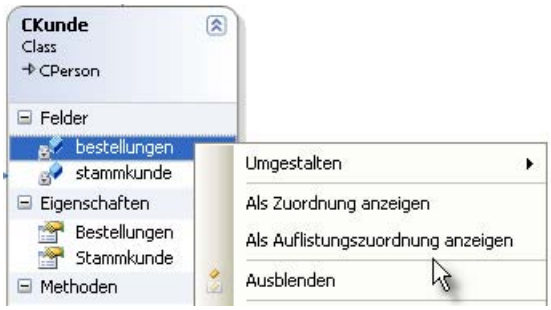


Fügen Sie zu *CKunde* ein Feld *stammkunde* vom *bool*-Datentyp hinzu und kapseln Sie es mit der entsprechenden Eigenschaft.

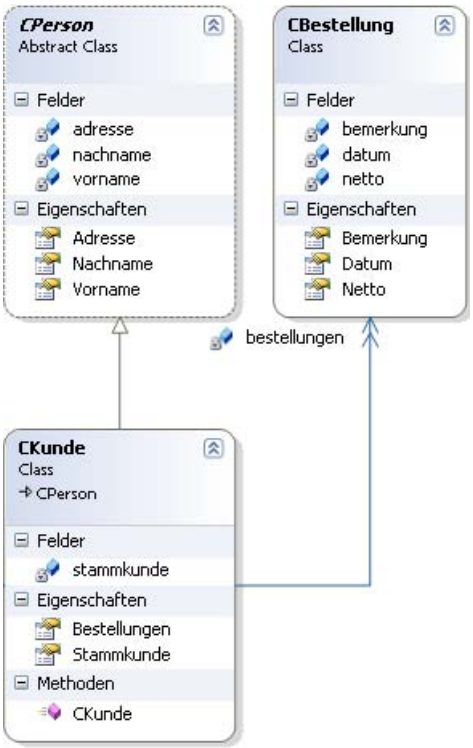
Etwas komplizierter wird es mit dem Feld *bestellungen*, welches ebenfalls hinzuzufügen ist. Der Datentyp ist *IList<CBestellung>*, also eine generische Liste vom Typ *CBestellung*.



Um die Assoziation zur Klasse *CBestellung* einzutragen, wählen Sie im Kontextmenü des Felds *bestellungen* den Eintrag *Als Auflistungszuordnung anzeigen*.



Im Ergebnis ist das Feld *bestellungen* verschwunden und stattdessen eine Auflistungszuordnung (multiple Assoziation) von der Klasse *CKunde* zur Klasse *CBestellung* im Klassendiagramm sichtbar (erkennbar auch an den doppelten Pfeilspitzen):



Wie Sie im obigen Klassendiagramm erkennen, haben wir zur Klasse *CKunde* noch eine Methode *CKunde* hinzugefügt. Wie es der Name bereits andeutet, handelt es sich hier um einen Konstruktor, den Sie ebenfalls über das Kontextmenü der Klasse hinzufügen:

Den Konstruktor brauchen wir, um beim Erzeugen eines Kunden-Objekts auch gleichzeitig eine (leere) Liste der Bestellungen zu erzeugen. Allerdings müssen wir diese Implementierung selbst vornehmen, da der Klassen-Designer nur Methodenrümpfe generieren kann:

```
public class CKunde : CPerson
{
        private IList<CBestellung> bestellungen;
        private bool stammkunde;
```

```
        public CKunde()
        {
```

Hier müssen wir selbst Hand anlegen:

```
            bestellungen = new List<CBestellung>();
        }

        public bool Stammkunde
        {
            get { return stammkunde; }
            set { stammkunde = value; }
        }

        public IList<CBestellung> Bestellungen
        {
            get { return bestellungen; }
            set { bestellungen = value; }
        }
}
```

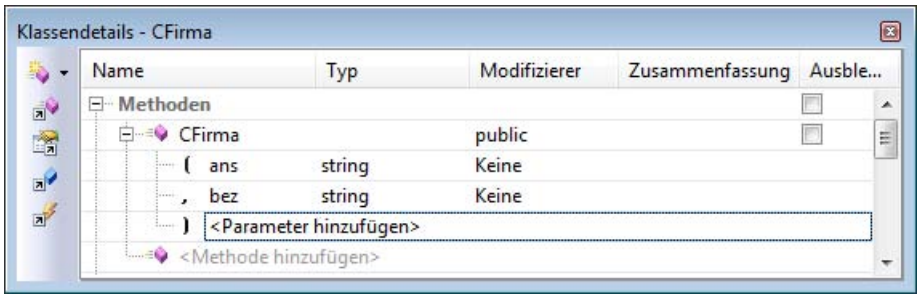
Klasse CFirma

Mittlerweile sind Sie beim Entwurf von Klassen so routiniert, dass nicht mehr erklärt werden muss, wie Sie die Klasse *CFirma* hinzufügen, mit den Feldern *anschrift* und *bezeichnung* ausstatten und diese durch entsprechende Eigenschaften kapseln.

Auch mit dem Feld *kunden,* dem Sie den Datentyp *IList<CKunde>* zuweisen, dürften Sie keine Probleme haben, denn die Vorgehensweise ist völlig analog zum Feld *bestellungen* der Klasse *CKunde*.

Die Assoziation zwischen den Klassen *CFirma* und *CKunde* wird erst dann in das Klassendiagramm eingezeichnet, wenn Sie dem Feld *kunden* per Kontextmenü den Eintrag *Als Auflistungszuordnung anzeigen* zuweisen. Am Quellcode selbst ändert sich dadurch nichts!

Die Klasse erhält einen eigenen Konstruktor (die Methode *CFirma*), diesem werden die Parameter für Bezeichnung und Anschrift der Firma übergeben. Den dazu erforderlichen Eintrag der Parameter *ans* und *bez* nehmen Sie im Fenster *Klassendetails* vor (Klick auf *<Parameter hinzufügen>*), alternativ könnten Sie das aber auch direkt im Code (siehe nachfolgendes Listing) vornehmen.



Wichtigste Aufgabe des Konstruktors ist das Erzeugen einer Instanz der Kundenliste, wobei die Implementierung von Ihnen im Codefenster ergänzt werden muss.

Der vollständige Code der Klasse *CFirma*:

```
public class CFirma
{
        private string bezeichnung;
        public string Bezeichnung
        {
            get { return bezeichnung; }
            set { bezeichnung = value; }
        }

        private string anschrift;
        public string Anschrift
        {
            get { return anschrift; }
            set { anschrift = value; }
        }

        private IList<CKunde> kunden;
        public IList<CKunde> Kunden
        {
            get { return kunden; }
            set { kunden = value; }
        }
```

Der Konstruktor:

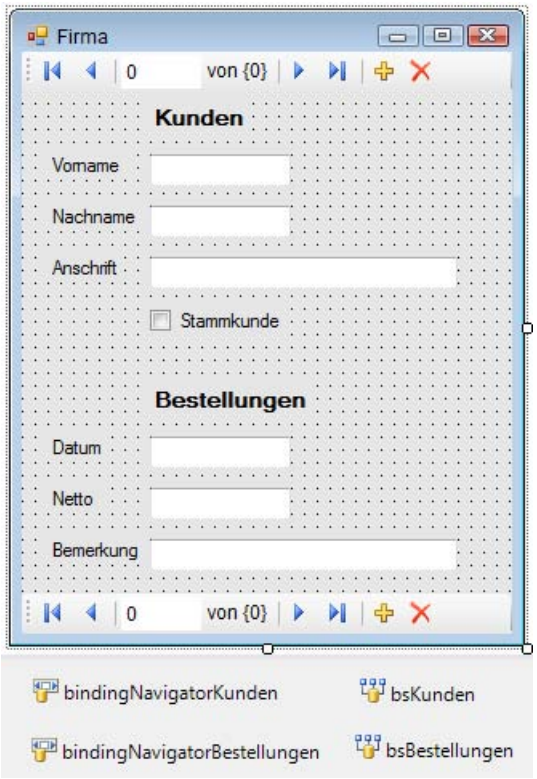
```
        public CFirma(string ans, string bez)
        {
```

Die Implementierung müssen Sie selbst vornehmen:

```
            bezeichnung = bez;
            anschrift = ans;
            kunden = new List<CKunde>();
        }
}
```

Klasse Form1

Die Geschäftslogik unseres Programms haben wir mit Hilfe des Klassendesigners fertig gestellt, was an Arbeit übrig bleibt, ist die Entwicklung einer geeigneten Benutzerschnittstelle auf Basis der bereits vorhandenen *Form1*. Neben sechs *TextBox*en einer *CheckBox* und einigen *Label*s benötigen wir für die Datenbindung zwei *BindingSource*- und zwei *BindingNavigator*-Komponenten.



Das Verbinden der beiden *BindingNavigator*-Komponenten mit ihrer Datenquelle erledigen wir gleich im Eigenschaftenfenster, indem wir die *BindingSource*-Eigenschaft auf *bskunden* bzw. *bsBestellungen* setzen.

```
public partial class Form1 : Form
{
```

Eine private Variable *firma* dient – wie wir später im Klassendiagramm noch sehen werden – als assoziative Verbindung zur Geschäftslogik (Klasse *CFirma*):

```
    private CFirma firma;
```

Im Konstruktor instanziieren wir die Geschäftslogik und verbinden die Steuerelemente mit der Kundenliste:

```
    public Form1()
    {
        InitializeComponent();
```

Instanz erzeugen:

```
        firma = new CFirma("4600 Altenburg, Fleischergasse 13", "DataBook");
```

Eine *BindingSource* kann neben der komplexen Datenbindung (*DataSet*) auch einfache Datenbindungen, wie die zu Auflistungen, verwalten:

```
            bsKunden.DataSource = firma.Kunden;
```

Steuerelemente anbinden:

```
            textBox1.DataBindings.Add(new Binding("Text", bsKunden, "Vorname", true));
            textBox2.DataBindings.Add(new Binding("Text", bsKunden, "Nachname", true));
            textBox3.DataBindings.Add(new Binding("Text", bsKunden, "Adresse", true));
            checkBox1.DataBindings.Add(new Binding("Checked", bsKunden, "Stammkunde", true));
        }
```

Bei Auswahl eines neuen Kunden wird das *CurrentChanged*-Ereignis der für die Kundenliste zuständigen *BindingSource* ausgelöst:

```
        private void bsKunden_CurrentChanged(object sender, EventArgs e)
        {
```

Der aktuelle Kunde kann über die *Current*-Eigenschaft der *BindingSource* gewonnen werden:

```
            CKunde kunde = (CKunde)bsKunden.Current;
```

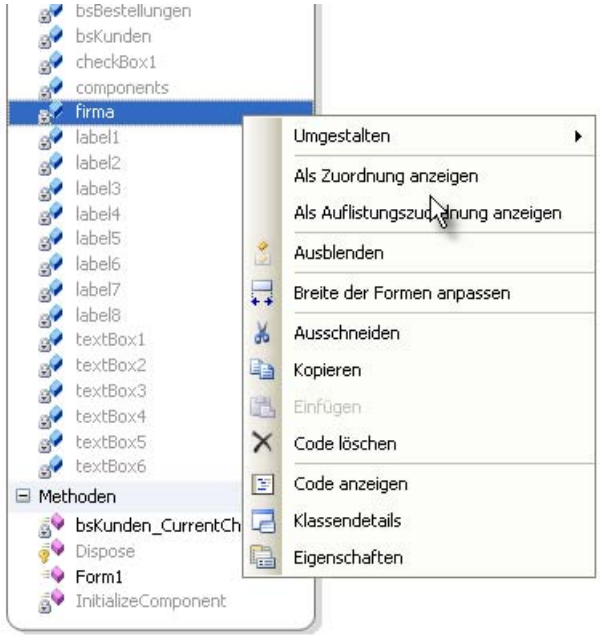
Jetzt ist auch der Zugriff auf die Bestellungen dieses Kunden möglich:

```
            bsBestellungen.DataSource = kunde.Bestellungen;
```

Da sich die Datenquelle geändert hat, müssen die Steuerelemente neu angebunden werden:

```
            textBox4.DataBindings.Clear();
            textBox4.DataBindings.Add("Text", bsBestellungen, "Datum", true);
            textBox5.DataBindings.Clear();
            textBox5.DataBindings.Add("Text", bsBestellungen, "Netto", true);
            textBox6.DataBindings.Clear();
            textBox6.DataBindings.Add("Text", bsBestellungen, "Bemerkung", true);
        }
}
```

Bevor wir mit dem Test der Anwendung beginnen, wollen wir noch das Klassendiagramm vervollständigen. Dazu brauchen wir aber die Klasse *Form1* nicht neu zu zeichnen (sie ist ja bereits vorhanden), sondern wir ziehen sie per Drag & Drop direkt vom Projektmappen-Explorer auf die Oberfläche des Klassen-Designers. Der Abschnitt "Felder" verwirrt mit einer unglaublichen Vielzahl von Einträgen, die vor allem von den beiden doch ziemlich komplexen *BindingNavigator*-Komponenten stammen. Unsere Aufmerksamkeit gilt aber vor allem dem *firma*-Feld, auf welches wir das Kontextmenü *Als Zuordnung anzeigen* anwenden.



Als Ergebnis erscheint im Klassen-Designer eine Assoziation zwischen den Klassen *Form1* und *CFirma*:

Sie liegen richtig wenn Sie vermuten, dass man im obigen Quellcode auf die explizite Deklaration der Variablen *firma* auch verzichten könnte und sie stattdessen als Feld im Klassen-Designer hinzufügt, der Designer würde die Variable dann automatisch in den Quellcode eintragen.

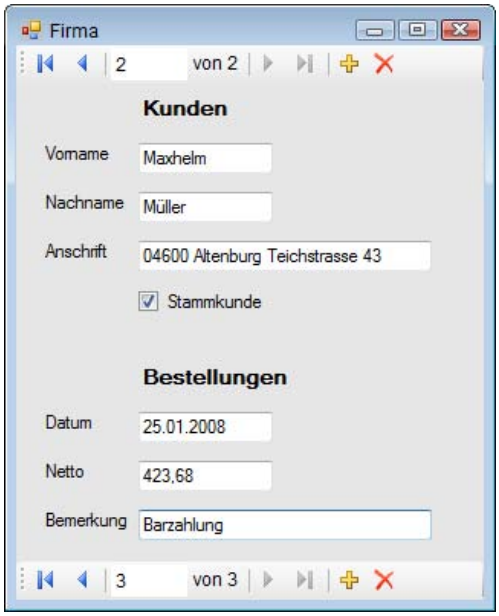
HINWEIS: Es ist eine Frage des persönlichen Geschmacks des Entwicklers, ob er Typ- oder Variablendeklarationen, Methodensignaturen etc. im Klassen-Designer oder im Quellcode vornimmt – die Änderungen werden sofort synchron auf die jeweils andere Sicht übertragen.

Abschlusstest

Nach Kompilieren und Starten der Anwendung erzeugen Sie zunächst einen oder auch mehrere Kunden. Klicken Sie also zuerst auf die "+" bzw. "Neu hinzufügen"-Schaltfläche des oberen *BindingNavigators*, ehe Sie die Eingabemaske füllen!

Zu jedem Kunden fügen Sie keine, eine oder mehrere Bestellungen hinzu. Auch hier klicken Sie zuerst die "Neu hinzufügen"-Schaltfläche des unteren *BindingNavigator*s, ehe Sie mit den Eintragungen beginnen.

Sie können beliebig in den Kunden und Bestellungen blättern und weitere Einträge hinzufügen bzw. löschen.



Das Programm arbeitet erstaunlich robust – der Klassen-Designer hat uns viel Arbeit abgenommen!

Bemerkungen

- Leider sind alle mühevoll eingegebenen Datensätze nach Beendigen des Programms futsch. Der Einbau einer zusätzlichen Persistenzschicht unter Verwendung serialisierter Objekte dürfte aber für den fortgeschrittenen Programmierer kein Problem sein.
- Um den Code nicht weiter aufzublähen wurde auf eine Fehlerbehandlung verzichtet.

Kapitel 26

Das Microsoft Event Pattern

Auf den Einsatz von Design Pattern (Entwurfsmuster) kann heute wohl kaum ein Softwareentwickler mehr verzichten. Das .NET Framework stellt dafür hervorragende Möglichkeiten auf der Basis von Schnittstellen, Delegates und Events zur Verfügung.

Im Folgenden wollen wir zunächst die theoretischen Grundlagen des bekannten *Observer-Pattern* erläutern, das innerhalb von .NET eine zentrale Rolle spielt.

Mit dem einfachen Beispiel einer Windows Anwendung "Zahlengenerator" sollen verschiedene Realisierungen demonstriert werden:

- klassische Implementierung auf Basis von Interfaces und Methoden-Callbacks,
- Implementierung auf Basis von Delegates und Events,
- MS-konforme Implementierung mittels Event Pattern.

HINWEIS: Eine erweiterte Anwendung mit sechs Event Pattern wird im Praxisbeispiel 26.9.1 gezeigt.

26.1 Einführung in Design Pattern

Design Pattern ermöglichen es, dass verschiedene Problemgruppen nach einem einheitlichen und erprobten Muster gelöst werden können, sie stellen einen Mechanismus zur Verfügung, mit dem bewährte Lösungen von anderen Entwicklern genutzt werden können.

Design Pattern wurden von der Softwareindustrie erstmalig 1995 eingeführt. Grundlage war das Buch *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Pub Co, 1995)* von Gamma, Helm, Johnson und Vlissides, in dem die Autoren insgesamt 23 Design Patterns für den Anwendungsentwurf vorstellten (C++, Smalltalk). Seitdem wurden diese Entwurfsmuster ständig weiterentwickelt und durch neue ergänzt.

Auch Microsoft hat in .NET viele der bekannten Entwurfsmuster eingesetzt und selbst beeindruckende Innovationen, wie z.B. das auf dem Delegate-Konzept beruhende Ereignismodell (Event Pattern), hinzugefügt.

Die Vorteile, die sich durch den Einsatz von Design Pattern ergeben, dürften auch den letzten Pattern-Muffel überzeugen:

- die Fehlerhäufigkeit sinkt, die Stabilität der Anwendung steigt,
- verbesserte Wartbarkeit und Wiederverwendbarkeit des Codes,
- unkomplizierte Erweiterung bestehender Anwendungen durch neue Klassen,
- Design Pattern sind nicht an eine bestimmte Sprache gebunden,
- die Verwendung einer einheitlichen Semantik verbessert die Kommunikation zwischen den Mitgliedern des Entwicklerteams,
- der Entwicklungsaufwand verringert sich teilweise dramatisch (Zeit ist Geld!).

Es dürfte klar sein, dass sich all diese Vorzüge umso deutlicher auswirken, je komplexer das Entwicklungsprojekt ist. Für die meisten Softwarefirmen ist deshalb der Einsatz von Design Pattern überlebenswichtig und ein Garant dafür, dass ihre über Jahre gepflegten Anwendungen nicht "aus dem Ruder laufen".

26.2 Aufbau und Bedeutung des Observer Pattern

Bei einem sauberen objektorientierten Entwurf sollte jedes einzelne Objekt nur für eine einzelne Aufgabe zuständig sein – mehr nicht! Nur diese Herangehensweise gewährleistet eine exakte Trennung zwischen den Objekten, was eine größere Wiederverwendbarkeit und Wartungsfreundlichkeit zur Folge hat.

Von größter Bedeutung ist eine solch klare Aufgabentrennung bei der Interaktion zwischen der Benutzerschnittstelle (User Interface) und dem darunter liegenden Geschäftsmodell, denn in der Entwicklungsphase kommt es häufig vor, dass sich das User Interface ändert, ohne dass ein entsprechendes Zusammenwirken mit dem Rest der Applikation gewährleistet ist.

Als Antwort auf diese Herausforderung wurden seit Einführung der GUI[1] eine Anzahl objektorientierter Frameworks entwickelt mit dem Ziel einer klaren Trennung zwischen User Interface und dem Rest der Applikation. Die meisten davon verwenden ein bestimmtes Design Pattern, dass diese Funktionalität zur Verfügung stellt.

Dieses Pattern ist unter Profis als *Observer Pattern* bekannt, seine Anwendung bezieht sich heute nicht nur auf die Abgrenzung zwischen Benutzerschnittstelle und Geschäftslogik, sondern überall darauf, wo eine saubere Trennung zwischen verschiedenen Objekten im System erforderlich ist[2].

26.2.1 Subjekt und Observer

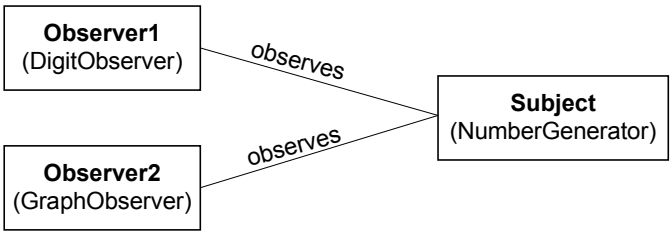
Obwohl es mehrere Variationen des Observer Pattern gibt, sind dabei grundsätzlich immer zwei Objektkategorien beteiligt: ein oder mehrere *Observer* und das *Subjekt*. Bezüglich User Interface

[1] Graphical User Interface

[2] Auch das bekannte MVC (*Model View Controler*)-Pattern lässt sich auf das Observer-Pattern zurückführen.

sind die Observer die Objekte, welche die Anzeige der Daten übernehmen. Andererseits repräsentiert das Subjekt einen bestimmten Teil der Geschäftslogik. Zwischen den Observern und dem Subjekt existieren logische Assoziationen. Sobald eine Änderung im Subjekt auftritt (z.B. die Modifikation einer Zustandsvariablen) beobachten die Observer diese Änderung und führen ein entsprechendes Update ihrer Anzeige aus.

Unsere Beispielapplikation soll Zufallszahlen generieren und sie in zwei Observer-Fenstern anzeigen. Der erste Observer übernimmt die zahlenmäßige Darstellung, der zweite Observer die Anzeige in Form eines einfachen Balkendiagramms:



Innerhalb unserer Applikation fungiert als Subjekt eine Instanz der Klasse *Model*, die das automatische Erzeugen der Zufallszahlen übernimmt. Diese Klasse kapselt eine Zustandsvariable, die der gerade aktuellen Zufallszahl entspricht. Um diese Information dem Benutzer in zwei unterschiedlichen Fenstern anzuzeigen, benutzt die Applikation zwei Observer: Eine Klasse *Observer1*, die das Ergebnis der Beobachtung in ein Label schreibt und eine Klasse *Observer2*, welche das Zeichnen des entsprechenden Balkendiagramms übernimmt. Beide Observer-Instanzen sind voneinander völlig entkoppelt und lassen sich zur Laufzeit beliebig beim Subjekt registrieren und deregistrieren.

HINWEIS: Weil der Generator ständig neue Zufallszahlen erzeugt, ändert sich auch seine Instanzenvariable entsprechend. Da die Observer das Subjekt ständig "beobachten", werden die Zustandsänderungen dem Benutzer quasi in "RealTime" angezeigt, d.h., sobald sie auftreten.

Dieser Beobachtungsprozess gewährleistet eine klare Abgrenzung zwischen *Model*-, *Observer1*- und *Observer2*-Klasse. Nehmen wir einmal an, dass sich die Anforderungen an die Applikation morgen ändern, beispielsweise wenn zusätzlich eine laufende statistische Auswertung der Zufallszahlen verlangt wird. Dank Observer-Pattern ist der Aufwand gering, lediglich eine neue Klasse *Observer3* wäre hinzuzufügen, die dann als dritter Observer fungieren würde. Die Klasse *Model* bleibt wie sie ist und "merkt" nicht einmal auf Anhieb, dass ein weiterer Observer angeschlossen wurde. Andererseits kann es sein, dass die *Model*-Klasse ihre Informationen von einer anderen Quelle holt, z.B. von einem Webdienst oder von einer Datenbank. Die Observer-Klassen bleiben wie sie sind und beobachten weiter unbeeindruckt ihr Subjekt.

26.2.2 Sequenzdiagramme

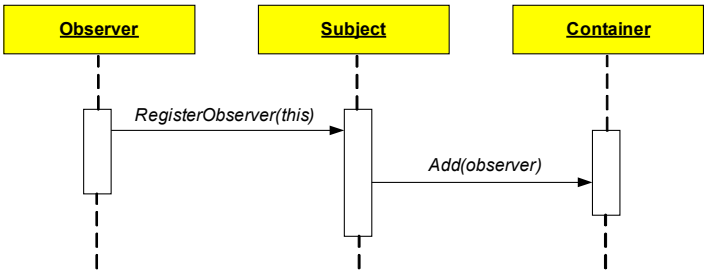
Wie bei den meisten Entwurfsmuster-Implementierungen steckt auch beim Observer-Pattern der Teufel im Detail. Falls – wie es zunächst nahe liegend scheint – unser *Model* die Observer instanzieren würde, gäbe es bedeutende Schwierigkeiten beim Anschluss weiterer Observer, da der Quellcode anzupassen und neu zu kompilieren wäre.

Der akkurate Weg – und damit die wichtigste theoretische Grundlage des Observer Pattern – besteht aus drei Etappen:

- *Registration*: Jeder Observer *registriert* sich beim Subjekt und drückt damit sein Interesse an einer Beobachtung aus.
- *Notification*: Sobald eine Zustandsänderung auftritt, *benachrichtigt* das Subjekt alle registrierten Observer über die gewünschten Änderungsdetails.
- *Unregistration*: Falls ein Observer nicht länger das Subjekt beobachten möchte, *deregistriert* er sich beim Subjekt.

Zur Illustration der drei Etappen eignen sich ganz ausgezeichnet die Sequenzdiagramme der UML, sie veranschaulichen allgemein die Interaktionen zwischen Instanzen im Zeitverlauf und besitzen in der Vertikalen eine zeitliche Dimension, in der Horizontalen werden die an der Interaktion beteiligten Objekte aufgeführt. Die senkrechte gestrichelte Linie verdeutlicht den Zeitverlauf und symbolisiert die Lebensdauer eines Objekts von seiner Entstehung bis zu seiner Entfernung. Senkrechte längliche Rechtecke, welche die gestrichelten Linien überlagern, geben an, welches Objekt gerade aktiv ist (Steuerungsfokus). Die waagerechten Pfeile entsprechen der Ausführung von Methoden.

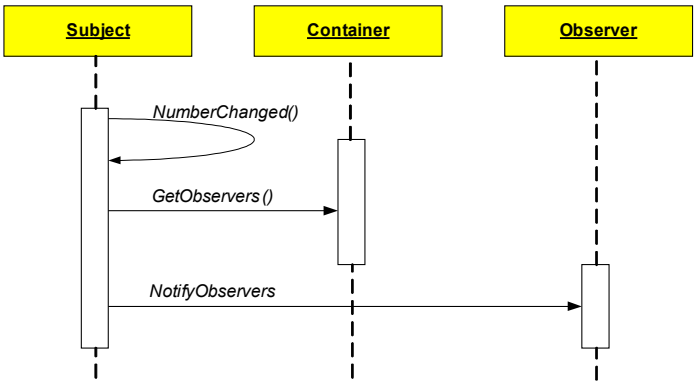
26.2.3 Die Registration-Sequenz



Der Observer ruft die *RegisterObserver*-Methode des Subjekts auf, wobei er sich selbst als Argument übergibt. Hat das Subjekt die Referenz erhalten, muss es diese abspeichern, um im Fall von Änderungen den (oder die) Observer benachrichtigen zu können. Anstatt aber die Observer-Referenz in einer Instanzenvariablen zu speichern, delegieren die meisten Observer-Implementierungen diese Verantwortung an ein separates Objekt, typischerweise einen Container, wie z.B. ein Array oder irgendeine andere Collection. Die Verwendung eines solchen Containers für das Speichern der Observer-Referenzen hat gewichtige Vorteile, auf die wir später noch einmal zurückkommen wollen.

Die abschließende Aktion ist das Abspeichern der Observer-Referenzen, ausgeführt durch die *Add*-Methode des Container-Objekts.

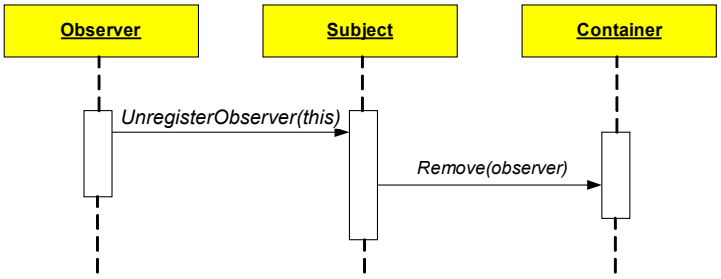
26.2.4 Die Notification-Sequenz



Sobald eine Zustandsänderung auftritt, holt sich das Subjekt alle im Container enthaltenen Observer (*GetObservers*-Methode).

Danach durchläuft das Subjekt alle Observer und benachrichtigt jeden einzelnen, indem es seine *NotifyObservers*-Methode ausführt, welche die Observer von der Zustandsänderung in Kenntnis setzt und dabei eine Objektvariable mit den Änderungsinformationen an die *UpdateDisplay*-Methode des jeweiligen Observers übergibt.

26.2.5 Die Unregistration-Sequenz



Diese Sequenz wird dann ausgeführt, wenn der Observer das Objekt nicht mehr länger beobachten möchte. Analog zur *Register*-Methode ruft der Observer jetzt die *UnregisterObserver*-Methode des Subjekts auf, wobei er sich selbst als Argument übergibt. Das Subjekt ruft dann die *Remove*-Methode des Containers auf, die den Observer entfernt.

26.2.6 Bedeutung der Sequenzen für das Geschäftsmodell

Während der Startphase der Anwendung registriert sich je eine Instanz der *Observer1*- und der *Observer2*-Klasse bei der *Model*-Instanz, wobei sich beide selbst als Argument an die *RegisterObserver*-Methode des *Model*-Objekts übergeben.

Das *Model*-Objekt legt beide *Observer*-Objekte in einem Container ab, das soll in unserem Demobeispiel eine generische *List* vom Typ *IObserver* sein:

```
private List<IObserver> observers = new List<IObserver>();
```

Alternativ käme z.B. auch eine *ArrayList* (veraltet) oder eine *Hashtable* in Frage. Sobald er eine neue Zahl generiert hat, benachrichtigt das *Model* die registrierten Observer durch Aufruf seiner *NotifyObservers*-Methode.

HINWEIS: Die *GetObservers*-Methode ist in unserem Fall nur fiktiv, sie wird in der konkreten Implementierung nicht benötigt und durch eine Schleife ersetzt, die die Felder der Liste durchläuft.

Auf Wunsch des Benutzers (oder beim Beenden der Applikation) deregistrieren sich beide Observer beim Modell (*UnregisterObserver*-Methode) und die Beziehung zwischen den Instanzen ist beendet.

26.2.7 Die Rolle des Containers

Bevor wir mit der Implementierung unseres Beispiels beginnen, lasst uns noch einmal die Vorteile beleuchten, die die Verwendung eines Containers anstatt von Instanzenvariablen mit sich bringt. Nehmen wir einmal an, Sie wollen einen dritten Observer einsetzen, z.B. den bereits erwähnten Statistik-Observer. Dazu entwickeln Sie eine neue Klasse *Observer3*, die der Anwendung problemlos hinzugefügt werden kann. Beim Starten der Applikation muss nun lediglich ein weiterer Observer registriert werden. Da das Subjekt viele Observer in seinem Container speichern kann, sind hier keinerlei Änderungen erforderlich (im Unterschied zur Verwendung von Instanzenvariablen). Wenn sich sein Zustand ändert, informiert die *NotifyObservers*[1]-Methode des Subjekts alle drei Observer. Sie sehen also, dass die Verwendung eines Containers ein äußerst flexibles Konzept darstellt, welches die Unterstützung mehrerer Observer durch ein Subjekt ermöglicht.

HINWEIS: Dem Subjekt ist es rein theoretisch erlaubt, unendlich viele Observer zu bedienen, ohne dass dazu nur eine einzige Zeile seines Quellcodes geändert werden müsste!

Viele Wege führen nach Rom. Um alle Zusammenhänge zu verstehen, programmieren wir im Folgenden unsere Applikation in drei Versionen, die systematisch aufeinander aufbauen und sich auf ihrem Weg zum Microsoft Observer Pattern immer weiter verbessern.

[1] Man beachte den "feinen" Unterschied bei der Namensgebung: *NotifyObservers*, weil **alle** Observer benachrichtigt werden. *RegisterObserver/UnregisterObserver,* weil sich jeweils nur **ein einziger** Observer an- bzw. abmeldet.

26.3 Implementieren mit Interfaces und Callbacks

Ein Interface (Schnittstelle) können Sie sich grob wie eine "leere" Klasse vorstellen, sie deklariert lediglich öffentliche Methoden, implementiert diese aber nicht. Letzteres wird von anderen Klassen erledigt, die die Schnittstelle verwenden und damit die Verpflichtung auf sich nehmen, **alle** ihre Methoden zu implementieren. Schnittstellen werden bekanntlich wie Klassen deklariert, können aber selbst nicht direkt instanziiert werden.

Zur Unterstützung der Implementierung von Observern und Subjekten werden hauptsächlich zwei Schnittstellentypen benötigt:

- Das *IObserver* Interface stellt die einzige Methode *UpdateDisplay* bereit. Dieses Interface wird von allen Klassen implementiert, die als Observer auftreten wollen.
- Das *IObservable* Interface wird von allen "beobachtbaren" Klassen, die als Subjekt fungieren, implementiert. Es liefert die bereits erwähnten Methoden *RegisterObserver*, *NotifyObservers* und *UnregisterObserver*.

HINWEIS: Die Verwendung dieser Schnittstellen hilft uns, die Kopplung zwischen den Observern und dem Subjekt auf ein Minimum zu reduzieren!

In unserem Beispiel agiert die *Model*-Klasse als Subjekt, als solches wird sie das *IObservable*-Interface implementieren. Entsprechend implementieren *Observer1*- und *Observer2*-Klasse das *IObserver* Interface.

Da alle Operationen durch das Interface fest vorgegeben sind und nicht durch eine spezielle Klasse, existiert keine feste Bindung zwischen dem Subjekt und seinen beiden Observern und umgekehrt. Dies erlaubt uns eine schnelle Änderung des spezifischen Observers oder des Subjekts, ohne den Rest der Applikation anzurühren.

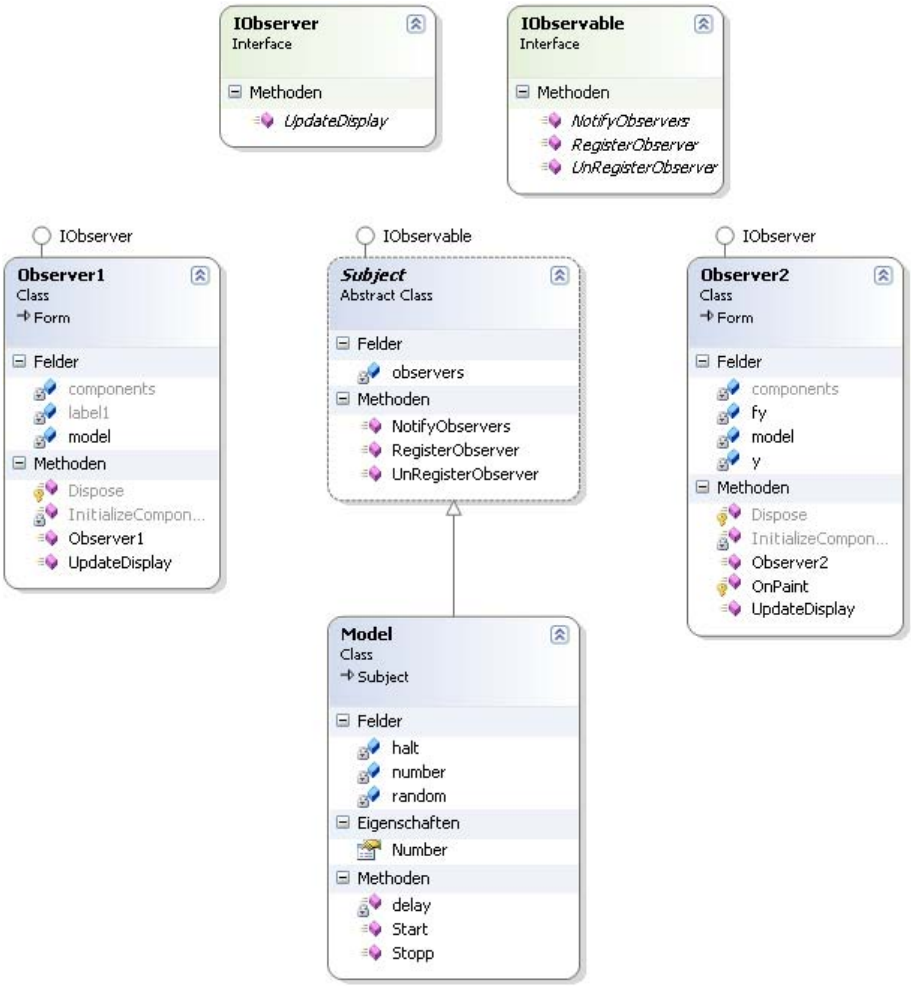
Zusätzlich zu den Interfaces wollen wir noch eine Basisklasse *Subject* zur Verfügung stellen, von der die Klasse *Model* erbt. Diese abstrakte Klasse steigert die Transparenz des Quellcodes und verringert den Implementierungsaufwand für das Observer Pattern vor allem dann, wenn mehrere Subjekte im Modell zu beobachten sind.

26.3.1 Übersicht und Klassendiagramm

Die nachfolgende Tabelle gibt eine Übersicht der im Projekt zu implementierenden Klassen:

Klasse	Beschreibung
Observer1	Formular, das die Klasse *Model* "beobachtet" und die Ziffernanzeige der erzeugten Zufallszahlen übernimmt.
Observer2	Formular, das die Klasse *Model* "beobachtet" und die grafische Anzeige der erzeugten Zufallszahlen übernimmt.
IObserver	Interface, das von den Klassen *Observer1* und *Observer2* implementiert wird und das die *UpdateDisplay*-Methode bereitstellt.

Klasse	Beschreibung
IObservable	Interface, das von der Klasse *Model* implementiert wird und das die Methoden *RegisterObserver*, *UnRegisterObserver* und *NotifyObservers* enthält.
Subject	Abstrakte Klasse, die das *IObservable*-Interface implementiert.
Model	Geschäftsmodell, das von der Klasse *Subject* erbt und das den Zahlengenerator implementiert.
Form1	Hauptformular, das Subjekt und Observer instanziiert und das die Registrierung/Deregistrierung beider Observer bei der Klasse *Model* ausführt.



Das abgebildete Klassendiagramm verdeutlicht die Zusammenhänge (ohne *Form1*). Es wurde mit dem Klassen-Designer von Visual Studio 2012 entworfen und leistete auch bei der Entwicklung des Quellcodes wertvolle Hilfe, da parallel zum visuellen Entwurf die Rumpfstrukturen der Klassen, Schnittstellen, Eigenschaften und Methoden automatisch generiert wurden.

HINWEIS: Beachten Sie, dass sowohl die Schnittstellen *IObserver* und *IObservable* als auch die abstrakte Klasse *Subject* allgemeinen Charakter tragen, also unverändert in beliebigen Anwendungen, die das Observer Pattern verwenden, eingesetzt werden können.

Wir starten Visual Studio 2und öffnen eine neue Windows Forms-Anwendung. Das Startformular *Form1* lassen wir zunächst unbeachtet liegen.

HINWEIS: Wer nicht auf die Unterstützung des Klassen-Designers verzichten möchte, sollte jetzt über das Menü *Projekt|Neues Element hinzufügen...* ein *Klassendiagramm* hinzufügen.

26.3.2 Die Schnittstellen IObserver und IObservable

Falls Sie ohne Klassen-Designer arbeiten, fügen Sie diese Schnittstellen als leere Klassen dem Projekt hinzu.

IObserver ist das Interface, welches von allen Observern implementiert wird:

```
public interface IObserver
{
        void UpdateDisplay(object o);
}
```

IObservable ist das Interface für alle observierten Klassen:

```
public interface IObservable
{
        void RegisterObserver(IObserver observer);
        void UnRegisterObserver(IObserver observer);
        void NotifyObservers(object o);
}
```

26.3.3 Die Abstrakte Klasse Subject

Von dieser Klasse, die *IObservable* implementiert, können alle zu beobachtenden Subjekte erben:

```
public abstract class Subject : IObservable
{
```

Eine generische Liste dient als Container für die Observer-Referenzen:

```
        private List<IObserver> observers = new List<IObserver>();
```

Wenn Sie mit dem Klassen-Designer gearbeitet haben, so hat dieser bereits fürsorglich für alle zu implementierenden Schnittstellenmitglieder eine Region angelegt:

```
        #region IObservable Member
```

Ein Observer wird zum Container hinzugefügt:

```
        public void RegisterObserver(IObserver observer)
        {
            observers.Add(observer);
        }
```

Ein Observer wird aus dem Container entfernt:

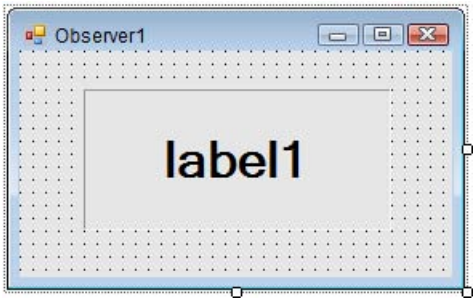
```
        public void UnRegisterObserver(IObserver observer)
        {
           observers.Remove(observer);
        }
```

Zur Benachrichtigung der registrierten Observer wird der Container durchlaufen. Innerhalb der Schleife erfolgt der Aufruf der *UpDateDisplay*-Methode eines jedes Observers, der ein beliebiges Geschäftsobjekt übergeben werden kann, welches die Informationen für die Aktualisierung der Anzeige enthält:

```
        public void NotifyObservers(object o)
        {
            foreach (IObserver obs in observers)
                obs.UpdateDisplay(o);
        }

        #endregion
}
```

26.3.4 Observer1

Ergänzen Sie das Projekt um eine weitere Windows Form, der Sie den Namen *Observer1* geben. Einzige Komponente der Oberfläche ist ein stattlich herausgeputztes Label.




```
public partial class Observer1 : Form, IObserver
{
        private Model model = null;

        public Observer1()
        {
```

```
            InitializeComponent();
        }
        #region IObserver Member
```

Die *UpdateDisplay*-Methode (siehe *IObserver*) gewinnt ihre Informationen aus der im Parameter übergebenen Objektreferenz. In unserem Fall ist das eine Instanz der *Model*-Klasse, aus der durch explizites Typecasting die *Number*-Eigenschaft "herausgezogen" wird:

```
        public void UpdateDisplay(object o)
        {
           model = (Model) o;
           this.Refresh();
          label1.Text = model.Number.ToString();
        }

        #endregion
}
```

26.3.5 Observer2

Analog zum Vorgänger ergänzen wir das Projekt um ein weiteres Formular mit dem Namen *Observer2*.

Die Oberfläche bleibt leer, umso umfangreicher wird stattdessen der Quellcode. Obwohl vom Prinzip her genauso aufgebaut wie sein Vorgänger, verschlingt das Erzeugen eines beweglichen Balkens doch einige Codezeilen mehr als die schlichte Anzeige im Label:

```
public partial class Observer2 : Form, IObserver
{
        private Model model = null;
        private int y = 0;
        private const int fy = 3;

        public Observer2()
        {
            InitializeComponent();
        }
```

Für Grafikoperationen ist das Überschreiben der *OnPaint*-Methode der Formular-Basisklasse immer einem *Form_Paint*-Eventhandler vorzuziehen[1]:

```
        protected override void OnPaint(PaintEventArgs e)
        {
            const int x1 = 120;        // linker Randabstand des Balkens
            const int y1 = 30;         // oberer Randabstand  "     "
            const int b = 30;          // Breite              "     "
            const int ymax = 50;       // maximale Höhe       "     "
            Graphics g = e.Graphics;
```

[1] Das ist eine Empfehlung von Microsoft, die auch verständlich wird, wenn man sich vorstellt, dass sich auch mehrere Eventhandler beim *Paint*-Ereignis registrieren können und die Reihenfolge ihrer Abarbeitung unbestimmt ist.

```
            g.FillRectangle(new SolidBrush(Color.Red), x1, fy * ymax + y1 - y, b, y);
            base.OnPaint(e);          // Aufruf der Basisklassenmethode nicht vergessen!
        }
```

Die folgende Schnittstellenmethode sieht (fast) so aus wie beim *Observer1*:

```
        #region IObserver Member
        public void UpdateDisplay(object o)
        {
            model = (Model) o;
            y = model.Number * fy;
            this.Refresh();
        }
        #endregion
}
```

26.3.6 Model

Diese Klasse kapselt das zu beobachtende Modell, sie erbt von der abstrakten Klasse *Subject*, die wiederum das *IObservable*-Interface implementiert hat. Die Klasse exportiert die Methoden *Start* und *Stopp* sowie die schreibgeschützte Eigenschaft *Number*, welche die aktuelle Zufallszahl repräsentiert.

```
public class Model : Subject
{
        private Random random = new Random();
        private int number = 0;
        private bool halt = false;
        public int Number
        {
            get
            { return number; }
        }
```

Zunächst eine Hilfsmethode für die Verzögerungszeit, die – im Unterschied zur *Sleep*-Methode des *System.Threading.Thread*-Objekts – ein Blockieren der Anwendung während der Pausen verhindert:

```
        private void delay(long zeit)
        {
            long zeit1 = System.Environment.TickCount;
            while ((System.Environment.TickCount - zeit1) < zeit) Application.DoEvents();
        }
```

Die Methode zum Starten des Zufallszahlengenerators:

```
        public void Start()
        {
            do
            {
                number = random.Next(50) + 1;     // Zufallszahl zwischen 1 und 50
```

Alle registrierten Observer werden benachrichtigt, die Übergabe der Änderungsinformationen erfolgt in der eigenen Klassenreferenz!

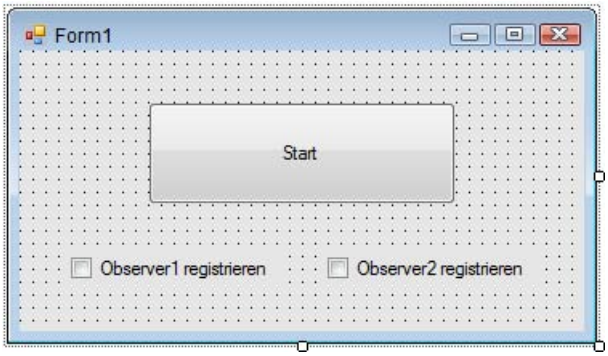
```
                this.NotifyObservers(this);
                this.delay(1000);
            }
            while (!halt);
        }
```

Schließlich noch die Methode zum Anhalten des Zufallszahlengenerators:

```
        public void Stopp()
        { halt = true;  }
}
```

26.3.7 Form1

Zum Schluss erledigen wir die Implementierung des bereits vorhandenen Hauptformulars *Form1*, das gewissermaßen die "Schaltzentrale" der Applikation ist. Neben der *Start*-Schaltfläche *Button1* werden zwei *CheckBox*en platziert, über die man beide Observer an- bzw. abmelden kann.



Der Code ist, dank Observer Pattern, so übersichtlich, dass die Erklärungen knapp gehalten werden können:

```
public partial class Form1 : Form
{
```

Das Subjekt und die beteiligten Observer werden referenziert:

```
        private Model model = null;
        private Observer1 observer1 = null;
        private Observer2 observer2 = null;
```

Die "Start"-Schaltfläche:

```
        private void button1_Click(object sender, EventArgs e)
        {
            checkBox1.Enabled = true;
```

```
            checkBox2.Enabled = true;
            button1.Enabled = false;
```

Alle benötigten Objekte werden instanziiert:

```
            model = new Model();
            observer1 = new Observer1();
            observer2 = new Observer2();
```

Und los geht's:

```
            observer1.Show();  observer2.Show();
            model.Start();
        }
```

Beide Observer können zur Laufzeit über die *CheckBox*en an- oder abgemeldet werden:

```
        private void checkBox1_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBox1.Checked)  model.RegisterObserver(observer1);
            else model.UnRegisterObserver(observer1);
        }

        private void checkBox2_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBox2.Checked)   model.RegisterObserver(observer2);
            else model.UnRegisterObserver(observer2);
        }
```

Vor dem Schließen des Formulars muss der Zahlengenerator angehalten werden, da bei laufender Schleife keine Ressourcenfreigabe erfolgen kann und Ihr Rechner irgendwo im Nirwana landen könnte:

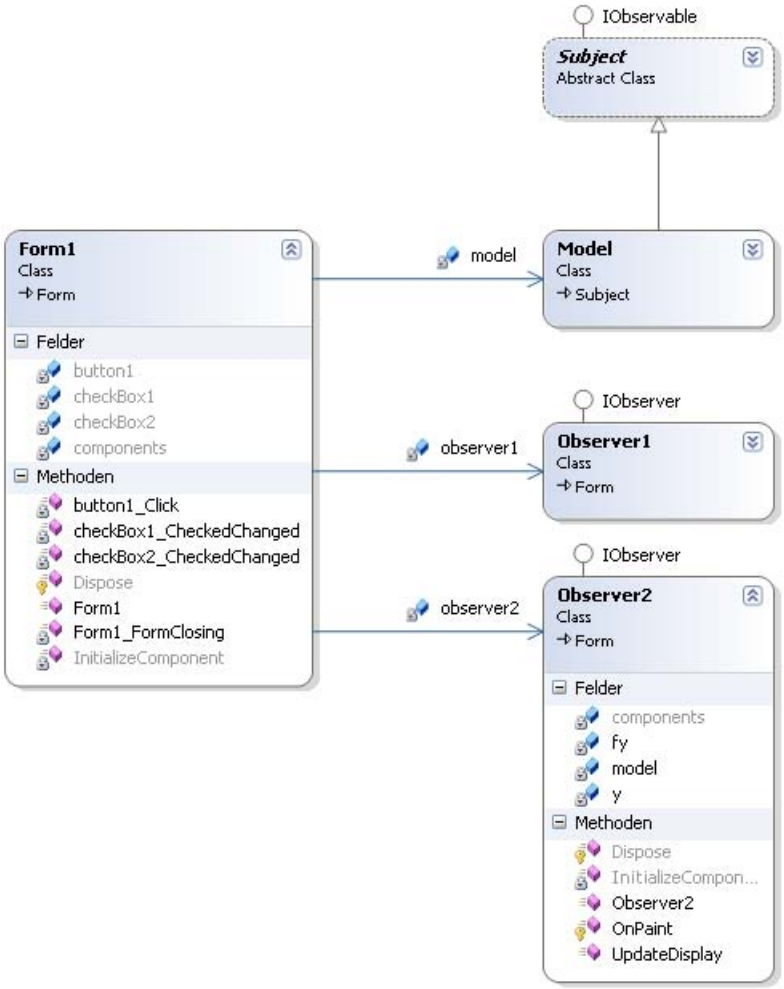
```
        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            model.UnRegisterObserver(observer1);
            model.UnRegisterObserver(observer2);
            model.Stopp();
        }
}
```

26.3.8 Ein zweites Klassendiagramm

Ein Projekt kann bekanntlich nicht nur durch ein, sondern auch durch mehrere Klassendiagramme dargestellt werden. Jedes davon liefert eine spezielle Sicht auf die Problemdomäne.

Da aus Gründen der Übersichtlichkeit die Klasse *Form1* nicht mit in unser erstes Klassendiagramm (siehe oben) aufgenommen wurde, holen wir dies jetzt nach und fügen einfach ein zweites Klassendiagramm zu unserem Projekt hinzu. Die Klasse *Form1* ziehen wir direkt aus dem Projektmappen-Explorer auf die Oberfläche des Klassen-Designers. Nach einigen zusätzlichen Aktionen, die hauptsächlich über das Kontextmenü der Klasse vorgenommen werden und mit denen wir weitere

in Beziehung stehende Klassen hinzufügen, erhalten wir das folgende recht aussagekräftige Diagramm:



26.3.9 Testen

Erst nach Klick auf die *Start*-Schaltfläche können Sie beide Observer registrieren. Da diese Reihenfolge wichtig ist, sind beide Checkboxen zu Beginn deaktiviert.

Wie Sie sehen, erfüllen beide Observer die ihnen zugedachte Aufgabe der Echtzeit-Beobachtung des Modells zur vollsten Zufriedenheit. Wenn Sie einen der beiden Observer aus der Registrierung des Subjekts entfernen, bleibt die Anzeige stehen und läuft erst weiter, wenn Sie die Registrierung erneuern.

26.4 Implementierung mit Delegates und Events

Da nun das Geheimnis des Observer Pattern gelüftet ist, wollen wir jetzt erkunden, ob das .NET-Framework vielleicht nicht ein noch effektiveres Implementierungsmodell für dieses Entwurfsmuster bietet. Stellt vielleicht das .NET-Framework Typen für *IObserver* und *IObservable*- Schnittstellen zur Verfügung? Nein, denn mit der Einführung von Delegates und Events stehen neue und leistungsfähigere Mechanismen für die Implementierung des Observer Pattern bereit. Äußerst bemerkenswert ist dabei, dass .NET selbst intensiven Gebrauch vom Observer Pattern macht und man deshalb dieses Pattern durchaus auch als das Grundkonzept des Delegate- bzw. Ereignismodells von .NET bezeichnen kann.

Ein Delegate ist bekanntlich ein objektorientiertes und typsicheres Äquivalent für einen Funktionszeiger (Pointer), weil dieser aufgrund seiner Risiken (Speicherzugriffsfehler!) unter .NET verpönt ist und nur noch in Ausnahmefällen geduldet wird. Eine Delegate-Instanz kapselt eine Referenz auf eine bestimmte Methode, wodurch ein anonymer Aufruf dieser Methode ermöglicht wird.

Hingegen repräsentiert ein Event ganz allgemein eine spezielles auf Klassenebene deklariertes Konstrukt, das zur Laufzeit zwecks Behandlung von Zustandsänderungen bestimmter Objekte eingesetzt wird.

26.4.1 Multicast-Events

Unter .NET spielen – basierend auf Multicast-Delegates – die Multicast-Events eine zentrale Rolle, sie erlauben es, eine theoretisch unbegrenzte Vielzahl verschiedener Eventhandler an ein einziges Ereignis zu binden Das .NET-Framework erzeugt zu diesem Zweck eine verlinkte Liste von Delegates und arbeitet diese nacheinander ab, sobald das Ereignis ausgelöst wird.

Vielleicht haben Sie die nun folgende fundamentale Erkenntnis schon im Voraus geahnt:

HINWEIS: Ein Multicast-Event ist quasi die formale Abstraktion der *RegisterObserver*, *UnRegisterObserver*- und *NotifyObservers*-Methoden!

Delegates werden zur Laufzeit bei einem bestimmten Ereignis registriert. Wenn das Ereignis ausgelöst wird, werden alle bei diesem Ereignis registrierten Delegaten aufgerufen und erhalten eine entsprechende Benachrichtigung von der aufrufenden Klasseninstanz.

Kehren wir zur Terminologie des Observer Pattern zurück, so ist das Subjekt die Klasse, die das Ereignis deklariert und auslöst, für unser konkretes Beispiel also *Model*. Was ändert sich also?

HINWEIS: Durch den Einsatz von Multicast-Events benötigt das zu beobachtende Modell das *IObservable*-Interface nicht mehr und muss auch nicht von der Basisklasse *Subject* erben. Lediglich ein Ereignis muss im Modell ausgelöst werden – mehr nicht!

Ähnliches gilt für die Observer-Klassen. Anstatt umständlich das *IObserver*-Interface zu implementieren und sich beim Subjekt zu registrieren, muss der Observer lediglich eine Delegate-Instanz erzeugen und diese mit dem entsprechenden Ereignis des Subjekts verbinden (die vom Observer erzeugte Delegate-Instanz muss natürlich zur Signatur der Ereignisdeklaration passen, ansonsten würde die Registrierung fehlschlagen).

HINWEIS: Ein Subjekt kann mehrere Observer durch das Auslösen eines bestimmten Ereignisses benachrichtigen. Die Observer registrieren/deregistrieren sich bei dem Ereignis, indem sie es mit einem Eventhandler verbinden bzw. von ihm trennen.

Für die nun folgende Implementierung der zweiten Version des Zahlengenerators nehmen Sie die folgenden Veränderungen an den Klassen/Schnittstellen unseres Projekts vor:

26.4.2 IObserver, IObservable und Subject

Die Schnittstellen *IObserver* und *IObservable* und die abstrakte Klasse *Subject* werden ersatzlos aus dem Projekt entfernt!

26.4.3 Observer1 und Observer2

Im Vergleich zur Vorgängerversion ändert sich fast nichts – lediglich auf die Implementierung der *IObserver* -Schnittstelle wird verzichtet.

26.4.4 Model

In dieser Klasse, die das Geschäftsmodell kapselt, sind einige wichtige Änderungen durchzuführen. Die Klasse ist selbstständig geworden, im Vergleich zur Vorgängerversion erbt sie nicht mehr von der (nun nicht mehr vorhandenen) abstrakten *Subject*-Klasse. Statt des Aufrufs von *NotifyObservers* kommt nun das .NET-Ereignismodell zum Zug:

```
public class Model
{
```

Grundlage ist ein von uns als *NumberDelegate* bezeichneter Methodentyp, der dem gewünschten Ereignistyp entspricht, d.h., er hat die Signatur der gewünschten Eventhandler bzw. der *Update*-Methode der Observer:

```
    public delegate void NumberDelegate(object o);
```

Ein Ereignis vom Typ des Delegaten ist zu instanziieren:

```
    public event NumberDelegate NumberChanged;
```

Nicht geändert hat sich die Implementierung des Zufallszahlengenerators:

```
    private Random random = new Random();
    private int number = 0;
    private bool halt = false;
...
```

Die *Start*-Methode hat sich gegenüber der Vorgängerversion dahingehend geändert, dass an die Stelle des Aufrufs von *NotifyObservers* der Aufruf der Ereignisinstanz tritt:

```
    public void Start()
    {
            do
            {   number = random.Next(50) + 1;
```

Das Auslösen des *NumberChanged*-Ereignisses (und damit die Benachrichtigung der angeschlossenen Observer) darf nur dann erfolgen, wenn mindestens ein Eventhandler angemeldet ist:

```
                if (this.NumberChanged != null) this.NumberChanged(this);
                this.delay(1000);
            }
             while (!halt);
    }
}
```

26.4.5 Form1

Spätestens hier dürfte Sie die Eleganz dieser Lösung überzeugen: An die Stelle des Aufrufs der *Register-/UnregisterObserver*-Methode der *Model*-Instanz tritt eine Verknüpfung des vom *Model* ausgelösten *NumberChanged*-Ereignisses mit den jeweiligen *UpdateDisplay*-Eventhandlern der beiden Observer.

Der Implementierungscode ist – bis auf das Registrieren der Ereignisbehandlungen – identisch zur Vorgängerversion:

```
...
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
        if (checkBox1.Checked)
                model.NumberChanged += new Model.NumberDelegate(observer1.UpdateDisplay);
        else model.NumberChanged -= new Model.NumberDelegate(observer1.UpdateDisplay);
}
```

Dank Multicast-Event darf sich auch ein zweiter Observer bzw. Eventhandler beim *NumberChanged*-Ereignis registrieren:

```
private void checkBox2_CheckedChanged(object sender, EventArgs e)
{
        if (checkBox2.Checked)
                model.NumberChanged += new Model.NumberDelegate(observer2.UpdateDisplay);
         else model.NumberChanged -= new Model.NumberDelegate(observer2.UpdateDisplay);
}
```

Der Abgesang:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
        if (model != null)
        {
            if (observer1 != null)
                   model.NumberChanged -= new Model.NumberDelegate(observer1.UpdateDisplay);
            if (observer2 != null)
                   model.NumberChanged -= new Model.NumberDelegate(observer2.UpdateDisplay);
        }
        model.Stopp();
}
```

26.4.6 Test und Vergleich

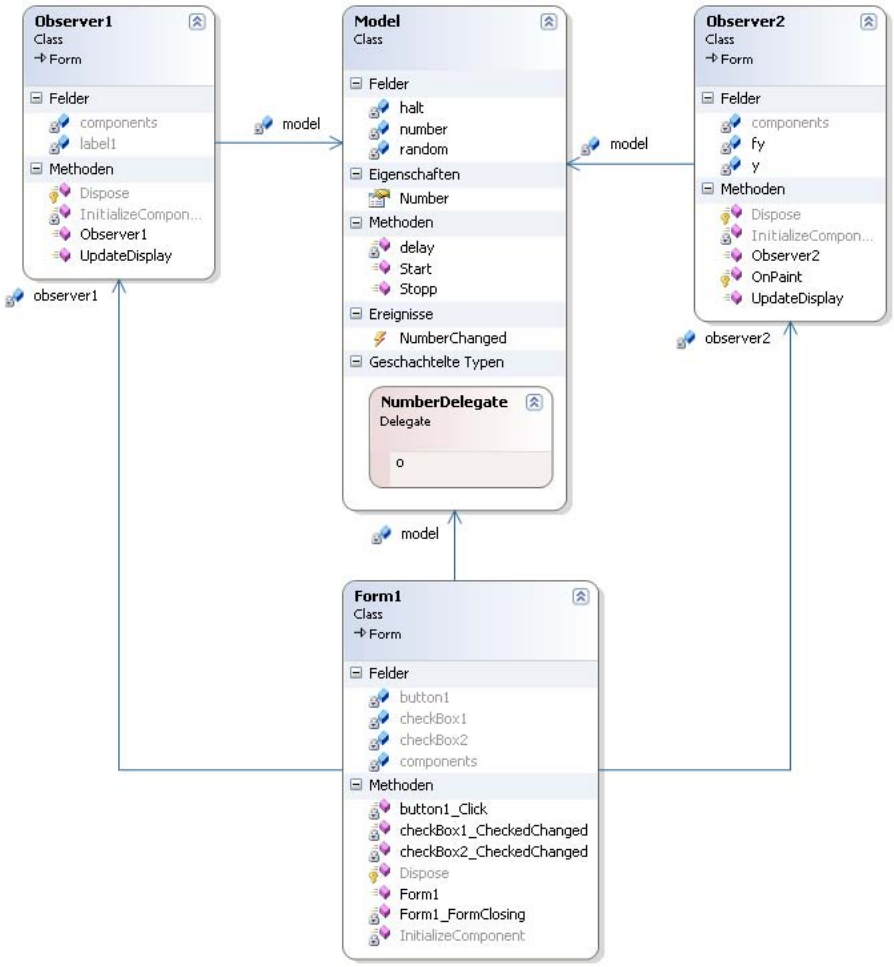
Beim Testen der Anwendung werden Sie keinerlei Unterschiede zur Vorgängerversion feststellen, die Vorzüge liegen also – wie nicht anders zu erwarten – allein auf der Seite des Programmentwicklers. Das heißt, der Quellcode ist kompakter und überschaubarer geworden, und die Klassen *IObserver*, *IObservable* und *Subject* haben sich in Luft aufgelöst.

HINWEIS: Hatte die erste Version noch den Vorzug, durch Einsatz der *IObserver*- und *IObservable*-Schnittstellen die Kopplung zwischen Observer und Subjekt auf ein Minimum zu reduzieren, so wird diese Kopplung in der vorliegenden Version durch Einsatz des .NET-Multicast-Ereignismodells komplett eliminiert!

Der größte Vorteil der Multicast-Ereignisdelegaten liegt aber darin, sich gleichzeitig auf verschiedene Methode in jeder Klasse Ihres Projekts beziehen zu können (natürlich müssen die Signaturen stimmen). Das befähigt jede Klasse, als Observer oder als Subjekt zu agieren (siehe Praxisbeispiel 26.9.1).

26.4.7 Klassendiagramm

Einen abschließenden Überblick über das Zusammenwirken der Klassen kann man sich auch hier durch das nachträgliche Erstellen eines Klassendiagramms verschaffen.



Natürlich kann man ein solches Diagramm auch schon von Anfang an in die Quellcode-Entwicklung einbeziehen, um sich durch das automatische Generieren von Klassen- und Methodenrümpfen die Arbeit zu erleichtern.

26.5 Implementierung des Microsoft Event Pattern

Basierend auf Events und Delegates wird das Observer Pattern von der .NET-Klassenbibliothek eifrigst strapaziert. Das .NET-Entwicklerteam hat die herausragende Bedeutung genau dieses Entwurfsmusters erkannt, da es sich sowohl auf UI-bezogene als auch auf nicht UI-bezogene Modelle anwenden lässt. Letztendlich entstand so eine leichte Modifikation des Observer Pattern, das *Event Pattern*.

Erwarten Sie aber keine allzu großen Überraschungen im Vergleich zur im vorhergehenden Abschnitt beschriebenen Variante, denn allgemein basiert dieses Pattern nur auf einer formalen Namenskonvention für Delegates, Events und Methodenaufrufe, die am Nachrichtenaustausch zwischen den verschiedensten Klasseninstanzen beteiligt sind. Obwohl dazu seitens der CLR keinerlei Zwang besteht, empfiehlt Microsoft trotzdem die strikte Verwendung dieses Pattern.

26.5.1 Namensgebung für Ereignisse

Die erste und wohl wichtigste Konvention bezieht sich auf die Namensgebung des vom Subjekt ausgelösten Ereignisses und sollte sich auf die Zustandsvariable bzw. Funktion beziehen, aufgrund der das Event "feuert".

Wir wollen dies anhand der *Model*-Klasse unseres Beispiels demonstrieren. Da das Ereignis durch eine Änderung der Zustandsvariablen *number* ausgelöst wurde, wäre der schlüssige Namen *NumberChanged*. Das hatten wir in der Vorgängerversion – auch ohne Kenntnis des Microsoft Event Pattern – bereits intuitiv richtig gemacht.

26.5.2 Namensgebung und Signatur der Delegates

Die zweite Konvention bezieht sich auf den Namen des Delegates und dessen Signatur. Der Name sollte sich aus dem Ereignisnamen entsprechend der ersten Konvention und dem Wort *Handler* zusammensetzen, für unser Beispiel also *NumberChangedHandler*.

Die Signatur des Delegates soll aus zwei Parametern bestehen, der erste heißt immer *sender* und ist eine Referenz auf das Objekt, welches das Ereignis auslöst. Der *sender*-Parameter ist immer vom Typ *System.Object* mit der Absicht, dass der Delegate potenziell an jede Methode eines jeden Objekts der Anwendung gebunden werden kann.

Der Namen des zweiten Parameters ist deutlich kürzer ausgefallen, einfach nur *e*, allerdings erfordert dieser Parameter ein wenig mehr Beachtung als es den Anschein hat: Er befähigt das Subjekt, willkürlich beliebige, meist mit dem Ereignis zusammenhängende, Daten an den Observer zu übermitteln. Falls eine solche Information nicht benötigt wird, genügt ein Platzhalter mit einer Instanz der Klasse *System.EventArgs*.

Anderenfalls muss eine Klasse von *System.EventsArg* abgeleitet werden, deren Implementierung dann die zu übergebenden Daten bereitstellt. Die Namensgebung setzt sich aus dem Namen des Ereignisses und dem Wort *EventArgs* zusammen, für unser Beispiel also *NumberChanged-EventArgs*.

26.5.3 Hinzufügen einer das Ereignis auslösenden Methode

In der Vorgängerversion hatten wir das Ereignis einfach innerhalb der *Start*-Methode der *Model*-Klasse aufgerufen. Das ist aber nicht im Sinne des Microsoft Event Pattern, denn dieses verlangt, dass das Auslösen eines Ereignisses durch eine spezielle Methode gekapselt werden muss. Der Name dieser Methode setzt sich aus dem Ereignisnamen und dem Präfix *On* zusammen. Die Sichtbarkeit dieser Methode sollte auf *protected* gesetzt werden. Bezogen auf unser *Beispiel* würde diese Methode also *protected void OnNumberChanged()* heißen.

Also noch einmal:

- *NumberChanged* ist das Ereignis.
- *OnNumberChanged* ist die Methode, in der das Ereignis ausgelöst wird.

Wenden wir uns nun der konkreten Implementierung zu. Da die Änderungen relativ gering ausfallen, brauchen Sie kein neues Projekt zu öffnen, sondern führen die Änderungen gleich am Vorgängerprojekt durch.

Neue Klasse NumberChangedEventArgs

Dieser Ereignisparameter ist für die Übertragung der vom Ereignis mitgegebenen Daten zuständig. Fügen Sie zum Projekt eine neue Klasse *NumberChangedEventArgs* hinzu:

```
public class NumberChangedEventArgs: EventArgs
{
        private object number;
```

Die Übergabe der Info erfolgt im Konstruktor, also gleich beim Erzeugen des Objekts:

```
        public NumberChangedEventArgs(object nr)
        {
            number = nr;
        }
```

Die ReadOnly-Eigenschaft *Number*:

```
        public int Number
        {
            get { return (int) number; }
        }
}
```

26.5.4 Model

Um die Konventionen des Microsoft Event Pattern einzuhalten, müssen hier vor allem Namensgebung bzw. Signatur von Delegate und Ereignis angepasst werden. Die neue Ereignisklasse *NumberChangedEventArgs* bewirkt in unserem Fall, dass die Implementierung der *Number*-Eigenschaft entfallen kann, da die Informationen nicht mehr im *object*-Parameter, sondern in der Instanz von *NumberChangedEventArgs* weitergereicht werden.

```
public class Model
{
```

Einen MS-konformen Ereignistyp deklarieren:

```
    public delegate void NumberChangedHandler(object sender, NumberChangedEventArgs e);
```

Ein MS-konformes Ereignis vom Typ des Delegaten instanziieren:

```
    public event NumberChangedHandler NumberChanged;
```

Der folgende Code bleibt in den meisten Passagen gegenüber der Vorgängerversion unverändert:

```
    private Random random = new Random();
    private int number = 0;
    private bool halt = false;

    private void delay(long zeit)
    {
        long zeit1 = System.Environment.TickCount;
        while ((System.Environment.TickCount - zeit1) < zeit) Application.DoEvents();
    }

    public void Start()
    {
        do
        {
            number = random.Next(50) + 1;
```

Das Auslösen des *NumberChanged*-Ereignisses wird in eine separate Methode *OnNumberChanged* verlagert:

```
            this.OnNumberChanged();     // !
            this.delay(1000);
        }
        while (!halt);
    }
```

Beim Aufruf des *NumberChanged*-Ereignisses werden zwei Parameter übergeben, wobei *this* dem *sender*-Parameter entspricht und die erzeugte Instanz von *NumberChangedEventArgs* dem Parameter *e*. Das Ereignis wird ausgelöst, wenn mindestens ein Eventhandler angemeldet ist. Achten Sie auf den *new*-Konstruktor, dem die neue Zufallszahl als Parameter, in einem *object*-Datentyp verpackt (Boxing!), übergeben wird:

```
        public void OnNumberChanged()
        {
            if (this.NumberChanged != null)  // mindestens ein Handler muss angemeldet sein!
                this.NumberChanged(this, new NumberChangedEventArgs(number));
        }

        public void Stopp()
        {  halt = true;  }
}
```

26.5.5 Observer1

Die Signatur der *UpdateDisplay*-Methode ist dem Microsoft-Standard anzupassen:

```
public partial class Observer1 : Form
{
        ...
        public void UpdateDisplay(object sender, NumberChangedEventArgs e)
        {
            this.Refresh();
            label1.Text = e.Number.ToString();
        }
    }
```

Die Codeänderungen für die Klasse *Observer2* sind völlig analog:

```
public partial class Observer2 : Form
{
        ...
        private int y = 0;
        private const int fy = 3;

        protected override void OnPaint(PaintEventArgs e)
        {
            Graphics g = e.Graphics;
            g.FillRectangle(new SolidBrush(Color.Red), 120, fy * 50 + y1 - y, 30, y);
            base.OnPaint(e);
        }

        public void UpdateDisplay(object sender, NumberChangedEventArgs e)
        {
            y = e.Number * fy;
            this.Refresh();
        }
    }
```

26.5.6 Form1

Bis auf die Änderung der Schreibweise des Ereignisdelegaten *NumberDelegate* in *NumberChangedHandler* sind weiteren Anpassungen erforderlich (der Namen des Ereignisses *NumberChanged* war zufällig richtig). Beispielhaft soll dies an der Registriermethode des ersten Observers gezeigt werden:

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
  if (checkBox1.Checked)
    model.NumberChanged += new Model.NumberChangedHandler(observer1.UpdateDisplay);
  else
    model.NumberChanged -= new Model.NumberChangedHandler(observer1.UpdateDisplay);
}
```

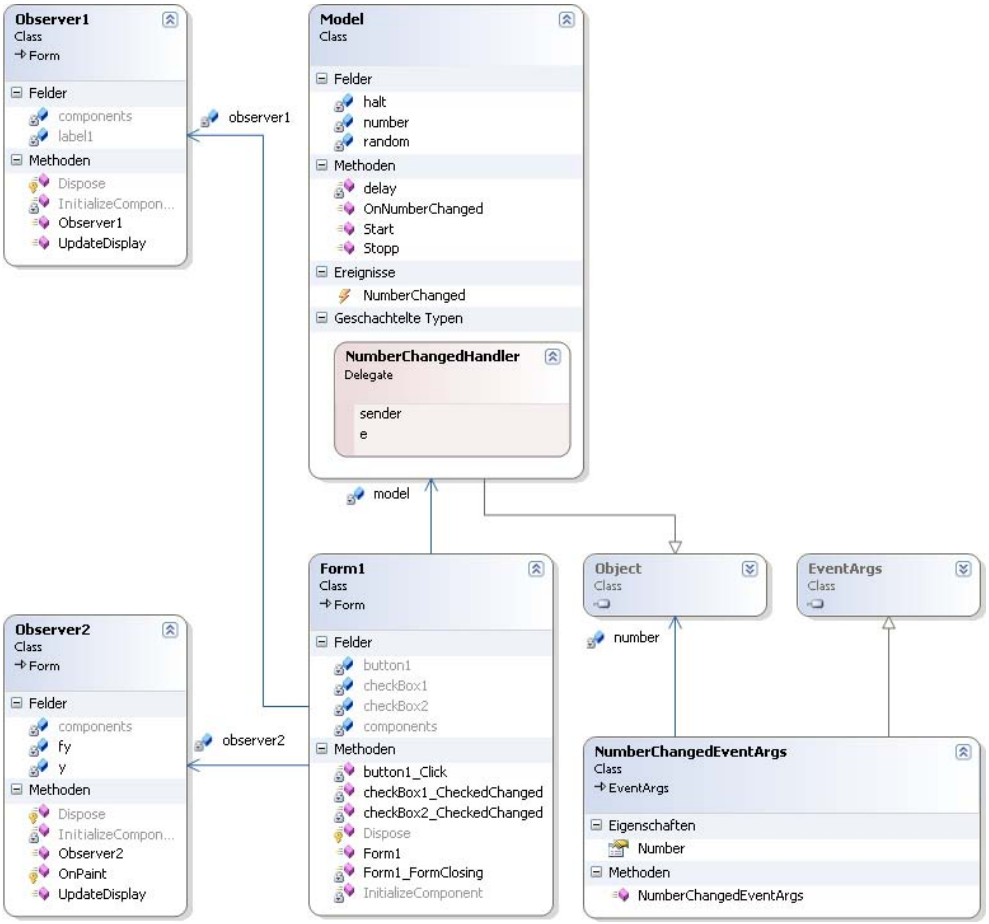
26.6 Test und Vergleich

Auch beim Test dieser Version gibt es erwartungsgemäß keinerlei Unterschiede zu den Vorgängerversionen. Zwar hat sich der Implementierungsaufwand geringfügig erhöht – es ist eine neue Klasse als Nachkommen von *EventArgs* hinzugekommen – aber die Vorteile des Microsoft Event Pattern dürften weitaus schwerer wiegen:

- Der Quellcode hat an Transparenz gewonnen, da einheitliche Konventionen eingeführt wurden.
- Lesbarkeit und Austauschbarkeit von Listings innerhalb des Entwicklerteams wurden deutlich verbessert.

26.7 Klassendiagramm

Interessante Erkenntnisse liefert auch hier ein vom Klassen-Designer erzeugtes Klassendiagramm. So fällt z.B. auf, dass – im Unterschied zur Vorgängerversion – beide Observer die Klasse *Model* nicht mehr direkt referenzieren (keine Assoziationsverbindung), denn die benötigte Information steckt im typisierten Übergabeobjekt *e*. Dies ist ein weiterer Beitrag zur Verringerung der Kopplung zwischen den Klassen.



26.8 Implementierung eines Event Pattern

Möglicherweise haben Sie in der Fülle des Quellcodes die Orientierung verloren, deshalb nochmals in Kürze die wichtigsten Schritte zur Implementierung eines Event Pattern am Beispiel des Ereignisses *NumberChanged*:

Definition einer von der Klasse *System.EventsArg* abgeleiteten Klasse *NumberChangedEventArgs*, dessen Konstruktor einen *object*-Parameter mit den zu übergebenden Informationen entgegennimmt und diese als typisierte ReadOnly-Eigenschaft (*Number*) bereitstellt.

```
public class NumberChangedEventArgs: EventArgs
{
        private object number;
        public NumberChangedEventArgs(object nr)
        {
            number = nr;
        }
```

```
        public int Number
        {
            get { return (int) number; }
        }
```

In dem das Ereignis auslösenden Subjekt (hier Klasse *Model*) deklarieren Sie einen Delegaten *NumberChangedEventHandler*, dessen Signatur den Typ des Ereignisses repräsentiert und definieren damit anschließend das Ereignis *NumberChanged*:

```
        public delegate void NumberChangedHandler(object sender, NumberChangedEventArgs e);

        public event NumberChangedHandler NumberChanged;
```

Fügen Sie eine Methode *OnNumberChanged()* hinzu, in welcher das Ereignis ausgelöst wird:

```
      public void OnNumberChanged()
      {
         if (this.NumberChanged != null)  // mindestens ein Handler muss angemeldet sein!
                this.NumberChanged(this, new NumberChangedEventArgs(number));
      }
```

Durch Aufruf von *OnNumberChanged()* innerhalb einer beliebigen anderen Methode des Subjekts werden die angeschlossenen Observer benachrichtigt.

In jedem Observer ist eine Methode (z.B. *UpdateDisplay()*) mit der Signatur des Delegaten *NumberChangedHandler* zu implementieren, in welcher die im Parameter *e* übergebenen Daten des Subjekts ausgewertet werden:

```
       public void UpdateDisplay(object sender, NumberChangedEventArgs e)
       {
            this.Refresh();
            label1.Text = e.Number.ToString();
       }
}
```

Ein Ereignis kann mit beliebig vielen Eventhandlern verbunden werden. Das Verbinden bzw. Registrieren eines Ereignisses mit einem Eventhandler erfolgt, indem man dem Ereignis eine neue Instanz des Ereignisdelegaten mit dem Namen des Eventhandlers als Parameter zuweist:

```
      model.NumberChanged += new Model.NumberChangedHandler(observer1.UpdateDisplay);
```

Analog das Abkoppeln bzw. Deregistrieren des Ereignisses:

```
      model.NumberChanged -= new Model.NumberChangedHandler(observer1.UpdateDisplay);
```

26.9 Praxisbeispiel

26.9.1 Objekte beobachten sich gegenseitig

Das Microsoft Event Pattern ist – wie jedes andere Pattern auch – weitgehend offen für Interpretationen. Hatten wir im Kapitel 26 dieses Pattern ausschließlich im strengeren Sinn angewendet (die Observer dienten nur der Beobachtung, was einem reinen Lesezugriff gleichkommt), so wollen wir jetzt zeigen, dass man das Pattern auch für Eingabezwecke verwenden kann.

Die Realisierung ist einfach, da Subjekt und Observer ihre Rollen vertauschen können, denn das Microsoft Event Pattern gilt ja ganz allgemein für Beziehungen zwischen beliebigen Klassen.

In unserem Beispiel soll ein Zufallszahlengenerator (Klasse *Model*) von zwei Observern "beobachtet" werden. *Observer1* zeigt die erzeugten Zufallszahlen in einem Label an, *Observer2* in einem Balkendiagramm. Diese Funktionalität bietet nichts Besonderes und wurde bereits im Abschnitt 26.4 beschrieben.

Neu ist, dass wir die Oberflächen der Klassen *Observer1* und *Observer2* um eine Eingabemöglichkeit für den Maximalwert der erzeugten Zufallszahlen ergänzen wollen (in der Version nach Abschnitt 26.4 war dieser Wert konstant 50). Das bedeutet, dass beide Observer ebenfalls "beobachtet" werden, und zwar vom Subjekt. Wenn in einem der beiden der Maximalwert geändert wird, löst dieser das *MaximumChanged*-Ereignis aus und das *Model* kann darauf reagieren, indem es den Bereich der erzeugten Zufallszahlen entsprechend ändert.

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung. Außer dem Startformular *Form1* fügen Sie noch zwei weitere Formulare zum Projekt hinzu: *Observer1* und *Observer2*.

Das Startformular *Form1* mit zwei *CheckBox*en dient zur Verwaltung der Anwendung (Instanzen erzeugen, Observer an- und abmelden). Um den Maximalwert der Zufallszahlen einzustellen, wurden zum *Observer1* eine *TextBox* und zum *Observer2* eine *TrackBar* hinzugefügt (siehe Laufzeitabbildung am Schluss des Beispiels). Zur Implementierung der drei Formulare kommen wir später.

NumberChangedEventArgs und MaximumChangedEventArgs

Entsprechend der Microsoft Konvention muss für jedes Ereignis eine von *System.EventArgs* abgeleitete Klasse definiert werden, deren Implementierung die zu übergebenden Daten bereitstellt. In unserem Beispiel haben wir es mit zwei Ereignissen zu tun:

- *NumberChanged* (tritt auf, wenn der Zahlengenerator eine neue Zufallszahl erzeugt hat) und
- *MaximumChanged* (tritt auf, wenn sich oberer Grenzwert der generierten Zahlen geändert hat).

Folgerichtig müssen wir zunächst die Klassen *NumberChangedEventArgs* und *MaximumChangedEventArgs* zum Projekt hinzufügen:

```
public class NumberChangedEventArgs: EventArgs
{
        private object number;

        public NumberChangedEventArgs(object nr)   // Konstruktor
        {  number = nr;  }

        public int Number      // Read-Only Eigenschaft
        {  get { return (int)number; }  }
}
```

Genau wie die Klasse *NumberChangedEventArgs* verfügt auch die Klasse *MaximumChanged-EventArgs* über eine durch eine schreibgeschützte Eigenschaft gekapselte private Zustandsvariable und einen Konstruktor mit einen Parameter, der die zu übermittelnden Informationen enthält:

```
public class MaximumChangedEventArgs : System.EventArgs
{
        private int max;

        public MaximumChangedEventArgs(object mx)  // Konstruktor
        {  max = (int) mx;  }

        public int Maximum         // Read-Only Eigenschaft
        {   get { return max; }  }
}
```

Model

Diese Klasse repräsentiert das Geschäftsmodell, welches einerseits von seinen Observern beobachtet wird, andererseits aber auch selbst die Observer beobachtet:

```
public class Model
{
        private Random random = new Random();    // Zufallszahl
        private int number = 0;       // aktueller Wert des Zahlengenerators
        private bool halt = false;    // Abbruchbedingung für Zahlengenerator
        private int max = 50;         // aktueller Maximalwert von number
```

Eine Hilfsroutine zur Zeitverzögerung:

```
        private void delay(long zeit)
        {
            long zeit1 = System.Environment.TickCount;
            while ((System.Environment.TickCount - zeit1) < zeit) Application.DoEvents();
        }
```

Startmethode des Zufallszahlengenerators:

```
        public void Start()
        {
            do
```

```
        {
            number = random.Next(max) + 1;
```

Ereignis auslösen und etwas warten:

```
            this.OnNumberChanged();
            this.delay(1000);
        }
        while (!halt);
    }
```

Sauberer Abbruch obiger Schleife:

```
    public void Stopp()
    {
        halt = true;
    }
```

Die Klasse *Model* löst die Ereignisse *NumberChanged* und *MaximumChanged* aus. Bevor aber ein Ereignis mit dem *event*-Schlüsselwort instanziiert werden kann, muss ein Delegate mit einer dem Microsoft Event Pattern entsprechenden Signatur deklariert werden. Das Ereignis selbst wird in einer entsprechenden *On...*-Methode ausgelöst:

```
    public delegate void NumberChangedHandler(object sender, NumberChangedEventArgs e);
    public event NumberChangedHandler NumberChanged;

    protected void OnNumberChanged()
    {
        if (this.NumberChanged != null)
            this.NumberChanged(this, new NumberChangedEventArgs(number));
    }
```

Völlig analog verfahren wir mit dem zweiten Ereignis:

```
    public delegate void MaximumChangedHandler(object sender, MaximumChangedEventArgs e);
    public event MaximumChangedHandler MaximumChanged;

    protected void OnMaximumChanged()
    {
        if (this.MaximumChanged != null)
            this.MaximumChanged(this, new MaximumChangedEventArgs(max));
    }
```

Die Klasse *Model* löst nicht nur obige zwei Ereignisse aus, mit denen beide Observer benachrichtigt werden, sondern muss umgekehrt auch noch beide Observer argwöhnisch "beobachten", d h., auf deren *MaximumChanged*-Ereignis reagieren:

```
    public void UpdateMaximum(object sender, MaximumChangedEventArgs e)
    {
```

Der Maximalwert des Zahlengenerators wird angepasst:

```
        max = e.Maximum;
```

Das *Model* darf auch nicht "vergessen", alle weiteren angeschlossenen Observer von der Änderung des Maximalwerts zu benachrichtigen (wenn z.B. in der *TextBox* von *Observer1* der Maximalwert geändert wurde muss auch *Observer2* darüber "Bescheid wissen", um die Position seiner *TrackBar* entsprechend zu korrigieren):

```
            this.OnMaximumChanged();
        }
}
```

Observer1

Diese Klasse löst in ihrer Rolle als Subjekt das Ereignis *MaximumChanged* aus und reagiert in der Rolle als Observer auf die Ereignisse *NumberChanged* und *MaximumChanged* der *Model*-Klasse:

```
public partial class Observer1 : Form
{
       ...
```

Ein eigenes Ereignis (*MaximumChanged*) hinzufügen:

```
        public delegate void MaximumChangedHandler(object sender, MaximumChangedEventArgs e);
        public event MaximumChangedHandler MaximumChanged;
```

Methode, in welcher das Ereignis ausgelöst wird:

```
        protected void OnMaximumChanged()
        {
            int max = Convert.ToInt32(textBox1.Text);
            if (MaximumChanged != null) MaximumChanged(this,new MaximumChangedEventArgs(max));
        }
```

Zwei fremde Ereignisse behandeln:

```
        public void UpdateDisplay(object sender, NumberChangedEventArgs e)
        {
            this.Refresh();
            label1.Text = e.Number.ToString();
        }

        public void UpdateMaxDisplay(object sender, MaximumChangedEventArgs e)
        {
            textBox1.Text = e.Maximum.ToString();
        }
```

Die Eingabe eines neuen Maximalwerts in die *TextBox* ist mittels Enter-Taste abzuschließen:

```
        private void textBox1_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox1.Text != String.Empty))
                try
                {
                   this.OnMaximumChanged(); // eigenes Ereignis auslösen
                }
```

```
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
}
```

Observer2

Der Code ist nur aufgrund der Grafikdarstellung des Balkendiagramms umfangreicher, bezüglich der Auslösung des *MaximumChanged*-Ereignisses und der Behandlung der *NumberChanged*- und *MaximumChanged*-Ereignisse der *Model*-Klasse herrscht völlige Analogie zum Code von *Observer1*:

```
public partial class Observer2 : Form
{
        ...
        private int y = 0;            // aktuelle Balkenhöhe
        private const int fy = 3;     // y-Verstärkungsfaktor (Pixel pro Wert Eins)

        protected override void OnPaint(PaintEventArgs e)
        {
            const int x1 = 120;     // linker Randabstand
            const int y1 = 30;      // oberer Randabstand (bei maximaler Balkenlänge)
            const int b = 30;       // Breite des Balkens
            const int ymax = 50;    //  maximale Länge des Balkens
            Graphics g = e.Graphics;
            g.FillRectangle(new SolidBrush(Color.Red), x1, fy * ymax + y1 - y, b, y);
            base.OnPaint(e);
        }
```

Ein eigenes Ereignis (*MaximumChanged*) deklarieren und auslösen:

```
        public delegate void MaximumChangedHandler(object sender, MaximumChangedEventArgs e);
        public event MaximumChangedHandler MaximumChanged;
```

Ereignismethode:

```
        protected void OnMaximumChanged()
        {
            int max = trackBar1.Value;
            if (MaximumChanged != null) MaximumChanged(this,
                new MaximumChangedEventArgs(max));
        }
```

Fremde Ereignisse behandeln:

```
        public void UpdateDisplay(object sender, NumberChangedEventArgs e)
        {
            y = e.Number * fy;
            this.Refresh();
        }
```

```
        public void UpdateMaxDisplay(object sender, MaximumChangedEventArgs e)
        {
            trackBar1.Value = e.Maximum;
        }
```

Internes UI-Ereignis behandeln:

```
        private void trackBar1_ValueChanged(object sender, EventArgs e)
        {
           this.OnMaximumChanged();  // eigenes Ereignis auslösen
        }
}
```

Form1

Das Hauptformular der Anwendung übernimmt das Instanziieren aller beteiligten Klassen und das Registrieren/Deregistrieren aller Ereignisse bei ihren Handlern:

```
public partial class Form1 : Form
{
        ...
        private Model model = null;
        private Observer1 observer1 = null;
        private Observer2 observer2 = null;
```

Start:

```
        private void button1_Click(object sender, EventArgs e)
        {
            checkBox1.Enabled = true;
            checkBox2.Enabled = true;
            button1.Enabled = false;
```

Geschäftsmodell instanziieren:

```
            model = new Model();
```

Das Objekt *observer1* erzeugen, relativ zu *Form1* positionieren und anzeigen:

```
            observer1 = new Observer1();
            observer1.StartPosition = FormStartPosition.Manual;
            observer1.Location =
               new Point(this.Left - observer1.Width + 140, this.Top - observer1.Height - 20);
            observer1.Show();
```

Gleiches gilt auch für das Objekt *observer2:*

```
            observer2 = new Observer2();
            observer2.StartPosition = FormStartPosition.Manual;
            observer2.Location = new Point(this.Left + 160, this.Top
                                        - observer2.Height - 20);
            observer2.Show();
```

Zufallszahlengenerator starten:

```
            model.Start();
        }
```

Das Objekt *observer1* an-/abmelden:

```
        private void checkBox1_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBox1.Checked)
```

Verknüpfung aller von *observer1* erzeugten und empfangenen Ereignisse mit ihren Handlern:

```
            {
                observer1.MaximumChanged += new
                   Observer1.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged += new
                   Model.NumberChangedHandler(observer1.UpdateDisplay);
                model.MaximumChanged += new
                   Model.MaximumChangedHandler(observer1.UpdateMaxDisplay);
            }
            else
            {
                observer1.MaximumChanged -= new
                    Observer1.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged -= new
                    Model.NumberChangedHandler(observer1.UpdateDisplay);
                model.MaximumChanged -= new
                 Model.MaximumChangedHandler(observer1.UpdateMaxDisplay);
            }
        }
```

Das Objekt *observer2* an-/abmelden:

```
        private void checkBox2_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBox2.Checked)
```

Verknüpfung aller von *observer2* erzeugten und empfangenen Ereignisse mit ihren Handlern:

```
            {
                observer2.MaximumChanged += new
                   Observer2.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged += new
                   Model.NumberChangedHandler(observer2.UpdateDisplay);
                model.MaximumChanged += new
                   Model.MaximumChangedHandler(observer2.UpdateMaxDisplay);
            }
            else
            {
                observer2.MaximumChanged -= new
                    Observer2.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged -= new
```

```
                Model.NumberChangedHandler(observer2.UpdateDisplay);
            model.MaximumChanged -= new
                Model.MaximumChangedHandler(observer2.UpdateMaxDisplay);
        }
    }
```

HINWEIS: An dieser Stelle lassen sich bei Bedarf weitere Observer an- bzw. abmelden, ohne dass die bestehenden Klassen geändert werden müssten!

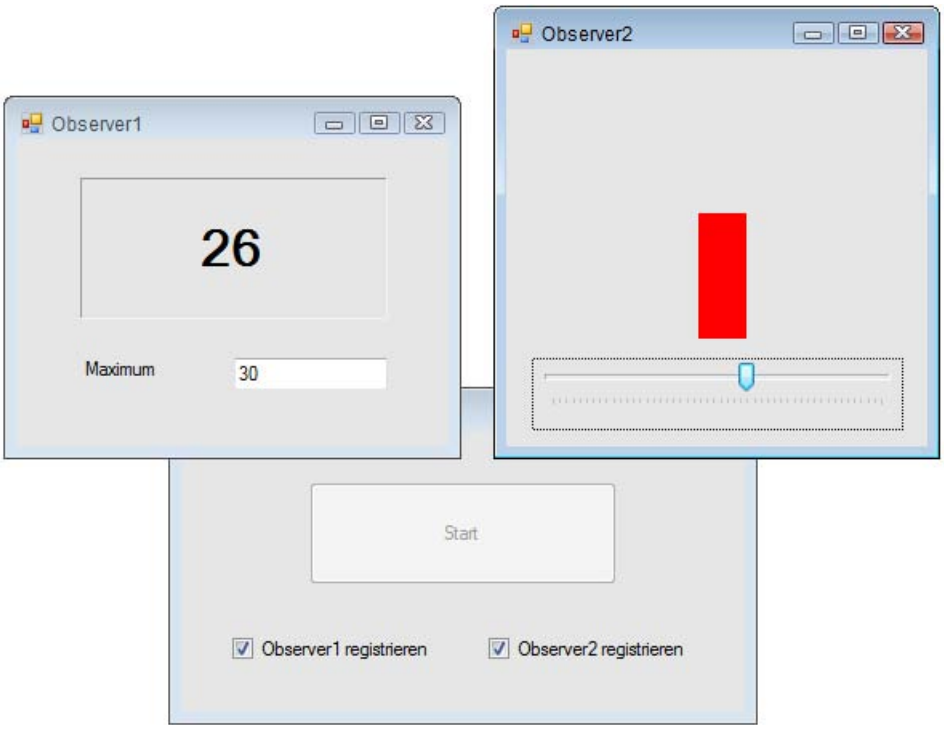
```
    // ...
```

Beim Schließen des Formulars werden alle Observer aus ihrer Registrierung entfernt und der Zahlengenerator wird angehalten:

```
    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        if (model != null)
        {
            if (observer1 != null)
            {
                observer1.MaximumChanged -=
                   new Observer1.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged -=
                   new Model.NumberChangedHandler(observer1.UpdateDisplay);
                model.MaximumChanged -=
                   new Model.MaximumChangedHandler(observer1.UpdateMaxDisplay);
            }
            if (observer2 != null)
            {
                observer2.MaximumChanged -=
                   new Observer2.MaximumChangedHandler(model.UpdateMaximum);
                model.NumberChanged -=
                   new Model.NumberChangedHandler(observer2.UpdateDisplay);
                model.MaximumChanged -=
                   new Model.MaximumChangedHandler(observer2.UpdateMaxDisplay);
            }
        }
        model.Stopp();
    }
}
```

Test und Vergleich

Nach Betätigen der "Start"-Schaltfläche registrieren Sie zunächst die beiden Observer. Wenn Sie einen neuen Maximalwert in die *TextBox* von *DigitObserver1* eingeben und anschließend die *Enter*-Taste drücken, springt wie von Geisterhand auch der Regler der *TrackBar* sofort auf seine neue Position und der Bereich, in welchem das *Model* seine Zufallszahlen erzeugt, verändert sich entsprechend.



Umgekehrt können Sie auch die *TrackBar*-Position direkt manipulieren, in der *TextBox* des *Observer1* erscheint sofort der neue Wert. In diesen beiden Fällen ist die Bezeichnung "Observer" eigentlich falsch, denn die Rollen von Observer und Subjekt sind hier ja vertauscht.

Da alle Objekte sauber voneinander entkoppelt sind, funktioniert die Anwendung absolut rückwirkungsfrei, der Ärger mit zirkulären Referenzen gehört der Vergangenheit an. Es dürfte absolut kein Problem sein, beliebige weitere Klassen hinzuzufügen, die miteinander interagieren.

Klassenzusammenstellung

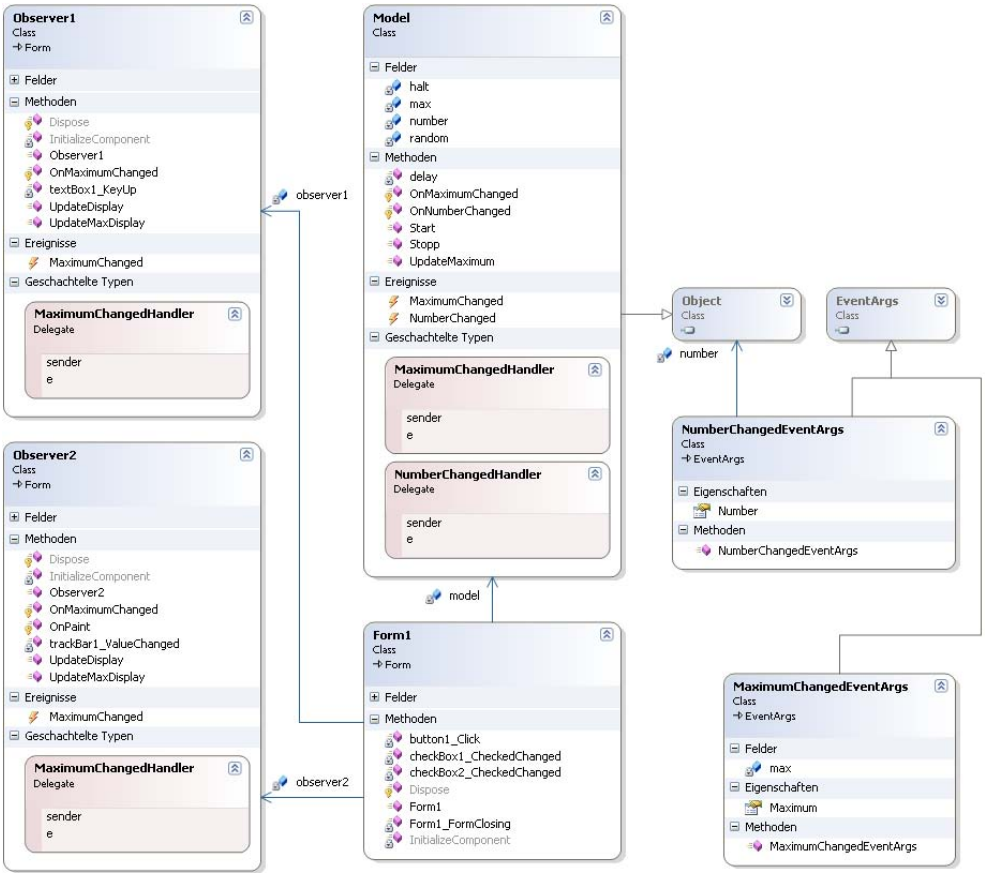
Die Tabelle gibt eine Übersicht aller wesentlichen Klassen des Objektmodells.

Klasse	Beschreibung
Form1	Hauptformular, welches die *Subject*- und *Observer*-Klassen instanziiert und welches die Registrierung/Deregistrierung aller Ereignisse ausführt.
Observer1	Formular, welches die Ziffernanzeige der erzeugten Zufallszahlen übernimmt und über welches man den Maximalwert der Zufallszahlen einstellen bzw. anzeigen kann.
Observer2	Formular, welches die grafische Anzeige der erzeugten Zufallszahlen übernimmt und über welches man den Maximalwert der Zufallszahlen einstellen bzw. anzeigen kann.

Klasse	Beschreibung
NumberChangedEventArgs	Klasse für das MS-konforme Argument des *NumberChanged*-Ereignisses.
MaximumChangedEventArgs	Klasse für das MS-konforme Argument des *MaximumChanged*-Ereignisses.
Model	Klasse, welche das Geschäftsmodell (ein Zufallszahlengenerator) kapselt.

Klassendiagramm

Da der Klassen-Designer in Visual Studio für uns immer "Gewehr bei Fuß" steht und mit geringstem Aufwand zu bedienen ist – warum sollten wir uns nicht einen abschließenden Blick auf das Klassendiagramm gönnen? Dadurch werden Zusammenhänge offengelegt, die beim alleinigen Hin- und Herblättern im Quelltext verborgen bleiben würden:



Kapitel 27

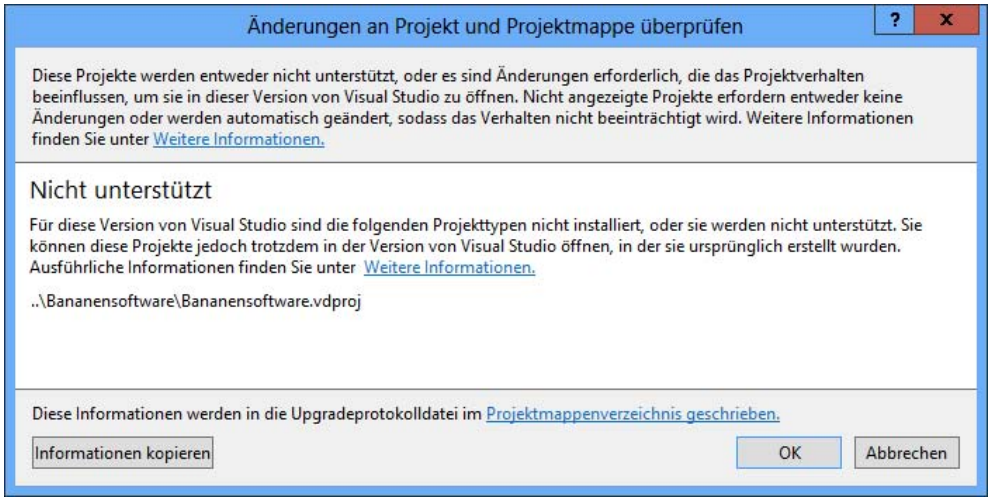
Verteilen von Anwendungen

Dem .NET-Programmierer bieten sich in Visual Studio 2012 zwei Hauptvarianten an, mit denen er seine Anwendung an den Endkunden[1] weitergeben kann:

- ClickOnce-Deployment
- InstallShield-Setup-Projekte

Während es sich beim ClickOnce-Deployment um eine eher junge Technologie handelt, bei der die Installationen über das Web abgerufen/aktualisiert werden können, ist die Verwendung von Setup-Projekten sicher jedem Windows-Nutzer bekannt.

Ganz nebenbei: Wer seine Altprojekte mit einem enthaltenen Setup-Projekt in Visual Studio 2012 öffnet, wird eine böse Überraschung erleben:



Die guten alten Microsoft Setup-Projekte werden also von Microsoft als "nicht mehr zeitgemäß" abqualifiziert und deshalb entsorgt. Ersatz sind jetzt die InstallShield-Setup-Projekte.

[1] Sehen wir einmal von den Distributionsmöglichkeiten für Webanwendungen ab.

27.1 ClickOnce-Deployment

Mit *ClickOnce* bietet sich dem .NET-Entwickler eine interessante Möglichkeit, seine Programme zu verteilen. Das Grundprinzip dieses Verfahrens ist es, das Verteilen und Aktualisieren von Anwendungen zu zentralisieren und damit auch zu vereinfachen.

27.1.1 Übersicht/Einschränkungen

Im Gegensatz zu den lokal installierten Windows-Anwendungen werden bei ClickOnce die Applikationen von einem zentralen Ort aus zur Verfügung gestellt. Dies darf eine Website, ein FTP-Server oder auch ein Dateipfad sein. Installiert bzw. ausgeführt werden kann die Anwendung von einer CD/DVD, einer Website oder einem UNC-Pfad.

HINWEIS: Sie entscheiden bereits beim Erstellen des Projekts, ob die ClickOnce-Anwendung nur online oder auch offline verfügbar ist. Im letzteren Fall wird dem Startmenü des Anwenders ein entsprechender Eintrag hinzugefügt.

Beim Herunterladen der Anwendungsdateien werden diese jedoch nicht wie gewohnt unterhalb des Verzeichnisses *\Programme* abgelegt, sondern im Benutzerprofil des aktuellen Nutzers (*\\Dokumente und Einstellungen\<Benutzer>\Lokale Einstellungen\Apps*).

HINWEIS: Aus obiger Tatsache resultieren auch einige Einschränkungen der ClickOnce-Distribution, so steht keine Möglichkeit zur Pfadauswahl zur Verfügung, auch die Installation selbst muss auf administrative Rechte/Aufgaben (Dateisystem, Registry) verzichten.

Ist die Anwendung auch offline verfügbar, bietet das ClickOnce-Verfahren die Möglichkeit, die Anwendungsdateien automatisch zu aktualisieren, ohne dass sich der Nutzer explizit darum kümmern muss.

Gesteuert wird das Verhalten der ClickOnce-Anwendung mit zwei Manifestdateien:

- Das Anwendungsmanifest beschreibt die für die Applikation erforderlichen Assemblies und weitere Dateien.
- Über das Bereitstellungsmanifest werden Versionsinformationen, Speicherorte und der Pfad des Anwendungsmanifests beschrieben.

HINWEIS: Die beiden Manifestdateien werden im Hintergrund vom Visual Studio-Assistenten, basierend auf Ihren Vorgaben, erstellt, Sie brauchen sich also nicht selbst darum zu kümmern.

Doch grau ist alle Theorie und so wollen wir Ihnen auch diesmal mit einem einfachen Beispiel die Schritte zum fertigen ClickOnce-Projekt aufzeigen.

27.1.2 Die Vorgehensweise

Visual Studio bietet zumindest zwei Wege, um das *ClickOnce*-Deployment vorzubereiten:

- Manuelles Einstellen der Deployment-Optionen, die Teil der Projekteigenschaften sind, auf den Registerkarten *Sicherheit* und *Veröffentlichen* und anschließendes Betätigen der Schaltfläche *Jetzt veröffentlichen...* auf der Registerkarte *Veröffentlichen*.
- Verwenden des Webpublishing-Assistenten durch Betätigen der Schaltfläche *Webpublishing-Assistent...* auf der Registerkarte *Veröffentlichen* oder Aufruf des Kontextmenüs *Veröffentlichen* im Projektmappen-Explorer.

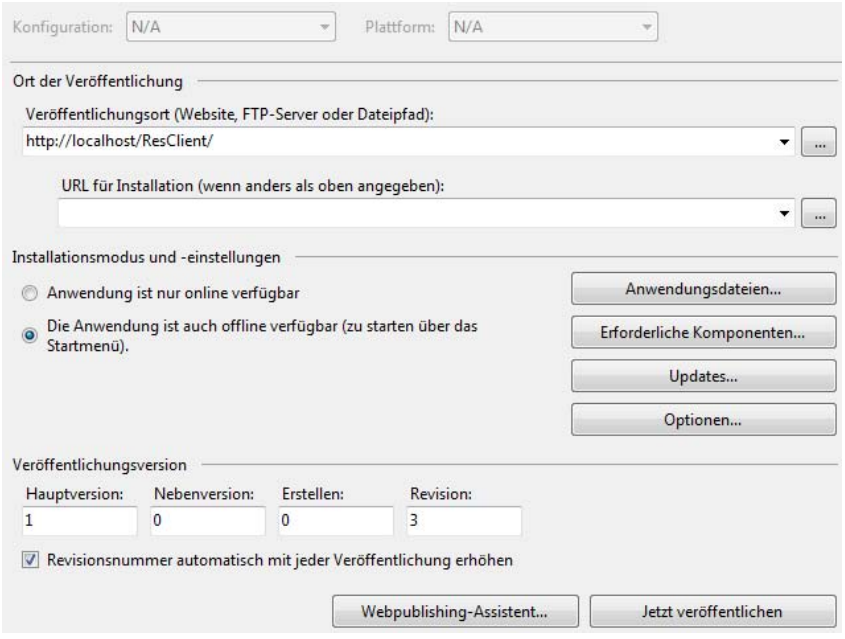
Im Folgenden wollen wir die Vorgehensweise am Beispiel einer aus zwei Komponenten bzw. Projekten (Client und Server) bestehenden Anwendung demonstrieren.

Da Sie zum *ClickOnce* Deployment keinerlei zusätzlichen Code schreiben müssen steht es Ihnen frei, stattdessen auch ein beliebiges anderes Projekt aus Ihrem Fundus zu verwenden. Allerdings sollte es nicht zu komplex sein, da während der Installation keinerlei Administratorenrechte zur Verfügung stehen und z.B. Einträge in die Registry nicht vorgenommen werden können.

Wir wollen zunächst den ersten der beiden oben genannten Wege beschreiten, also auf den direkten Einsatz des Webpublishing-Assistenten verzichten.

27.1.3 Ort der Veröffentlichung

Rufen Sie das Menü *Projekt|ResClient*-Eigenschaften... des Projekts *ResClient* auf und wählen Sie die Registerseite *Veröffentlichen*:

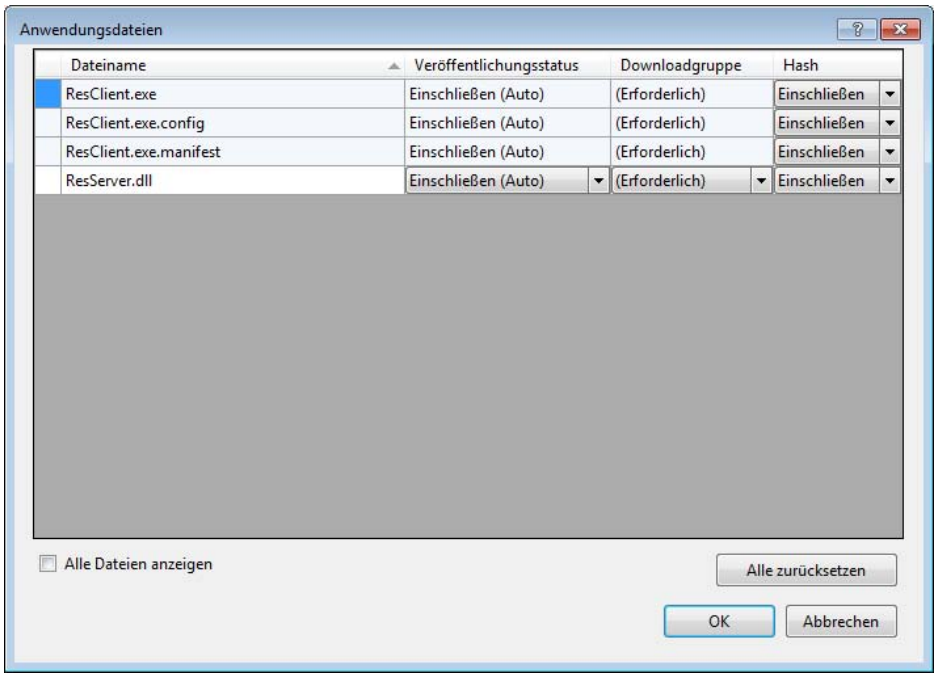


Tragen Sie dort den Ort der Veröffentlichung ein, für einen ersten Test eignet sich der lokale Webserver (IIS), alternativ können Sie auch einen lokalen Pfad oder einen FTP-Server angeben.

HINWEIS: Die Registerseiten *Sicherheit* und *Veröffentlichen* werden Sie beim *ResServer*-Projekt vergeblich suchen, da es sich hier um keine für das Deployment geeignete *(*.exe*-) Anwendung handelt.

27.1.4 Anwendungsdateien

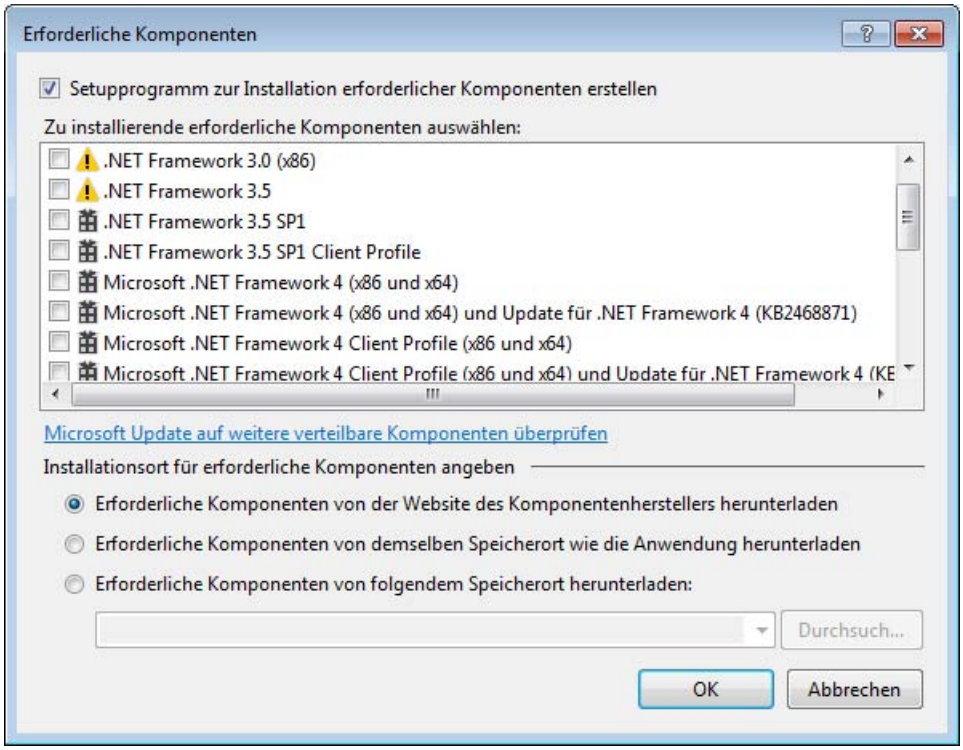
Klicken Sie auf die Schaltfläche *Anwendungsdateien...*, so öffnet sich ein Dialog, in dem Sie angeben können, welche Dateien auf den Deployment-Server zu kopieren sind. *ClickOnce* erkennt selbstständig, welche Abhängigkeiten zwischen den Dateien bestehen und welche Dateien unbedingt erforderlich sind. Seien Sie also unbesorgt, falls die für unser Beispiel erforderliche *ResServer.dll* nicht mit in der Liste aufgeführt ist.



27.1.5 Erforderliche Komponenten

Nach Klick auf die Schaltfläche *Erforderliche Komponenten...* zeigt Ihnen ein Dialog die Komponenten an, die bei Bedarf ebenfalls mit auf den Deployment-Server kopiert werden können.

Neben dem *.NET Framework 2, 3, 3.5, 4, 4.5* stehen momentan der *Microsoft Report Viewer, der SQL Server 2012 Express/LocalDB, der Windows Installer 4.5* und einige andere zur Verfügung. Für unser einfaches Beispiel können wir aber auf zusätzliche Komponenten verzichten.



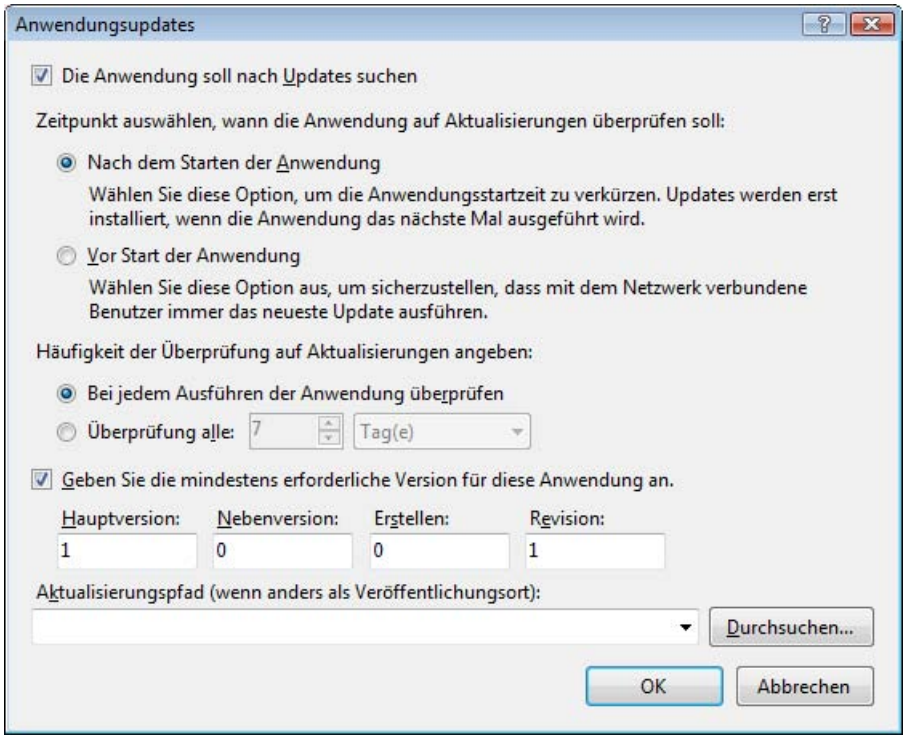
27.1.6 Aktualisierungen

Ein entscheidendes Merkmal der *ClickOnce*-Technologie ist das Update-Management, welches auf Wunsch auch vollautomatisch ablaufen kann.

Nach dem Klick auf die *Updates...*-Schaltfläche öffnet sich der entsprechende Dialog (siehe folgende Abbildung), der – wie übrigens alle anderen auch – so gut beschriftet ist, dass zusätzliche Erklärungen an dieser Stelle fast schon Papierverschwendung wären.

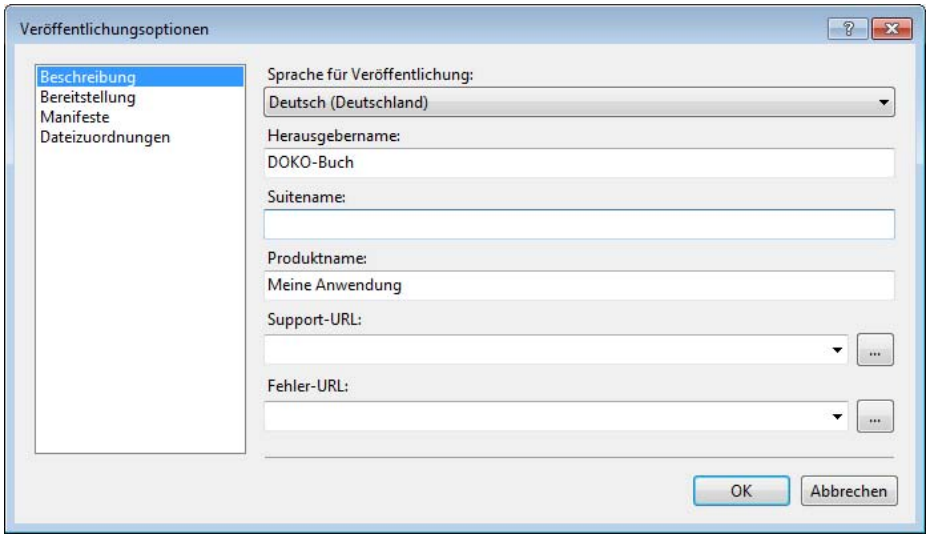
Sie entscheiden, ob (und wenn ja wann) auf Updates geprüft werden soll (vor oder nach dem Start der Anwendung). Bedenken Sie, dass Internet-Verbindungen nicht immer rasend schnell sein müssen, meist reicht es auch aus, nach dem Start zu prüfen und beim nächsten Programmstart steht die neue Version zur Verfügung.

Weiterhin können Sie das Update-Intervall, d h. die Häufigkeit der Updatesuche, bestimmen.



27.1.7 Veröffentlichungsoptionen

Legen Sie hier die wesentlichen Daten zu Ihrem Setup-Projekt fest (Herausgeber, Bereitstellungsseite, Dateizuordnungen etc.):

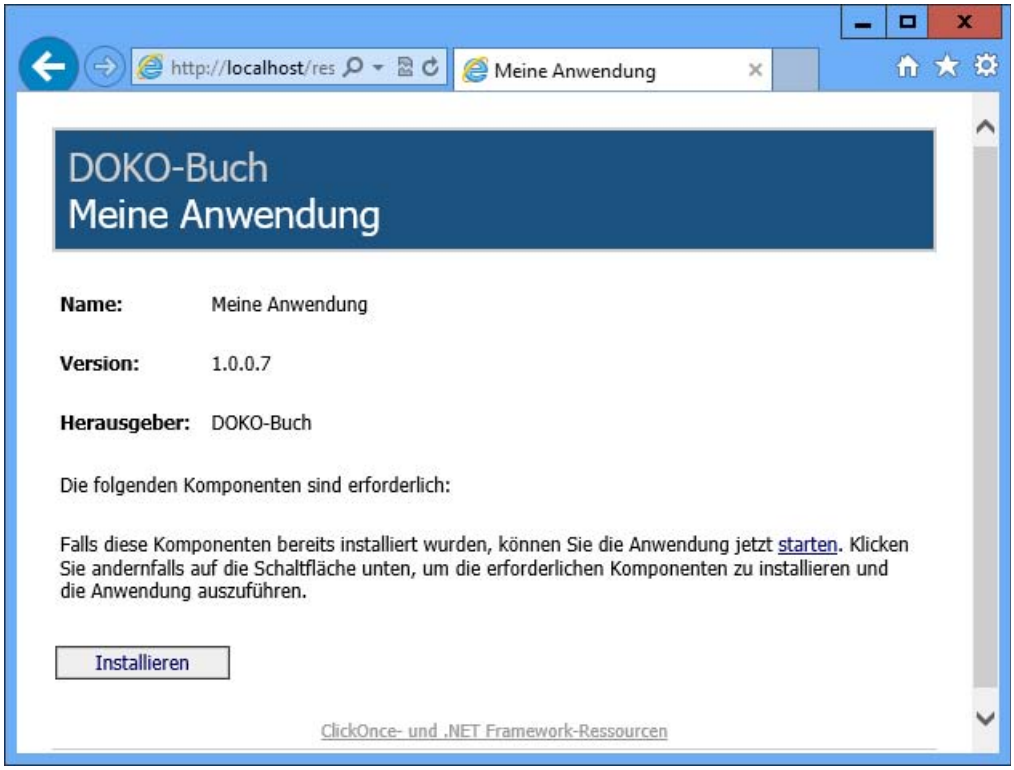


27.1.8 Veröffentlichen

Haben Sie alle Einstellungen vorgenommen, so steht dem entscheidenden Klick auf die Schaltfläche "Jetzt veröffentlichen" nichts mehr im Wege.

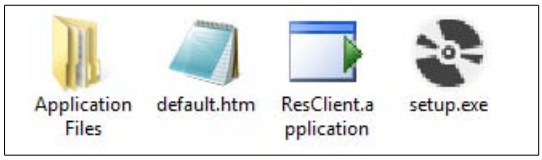
Es vergeht etwas Zeit, bis im Internet-Explorer die gewählte HTML-Seite (*default.htm*) erscheint, welche zusammen mit den Programmdateien auf dem Webserver liegt.

Die Installationswebsite:



27.1.9 Verzeichnisstruktur

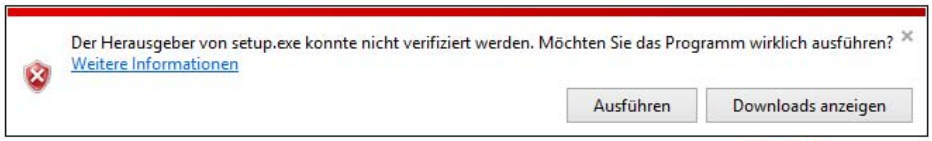
Auf dem Webserver wird das Verzeichnis *\ResClient* mit der abgebildeten Struktur angelegt:

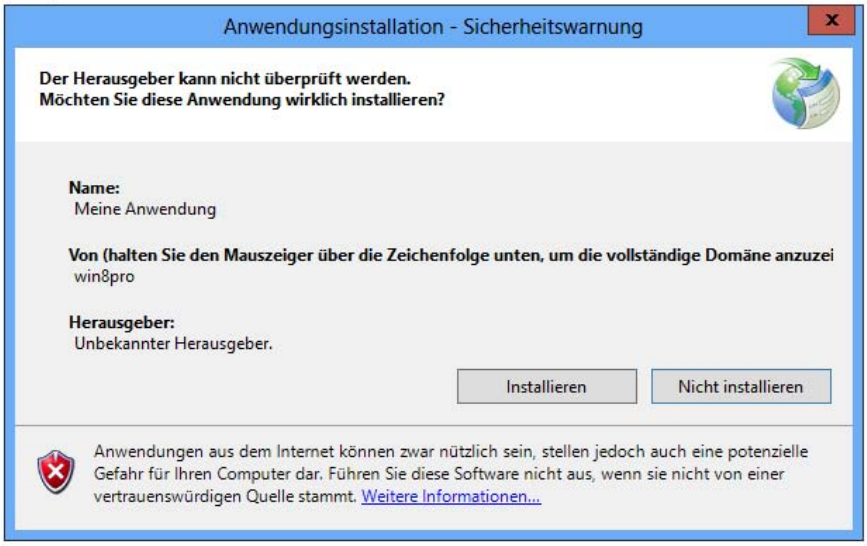


Ein paar Worte zu den einzelnen Dateien:

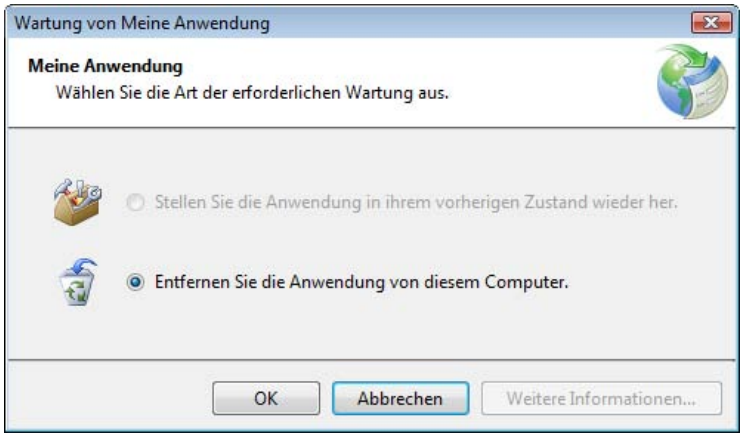
- Die Datei *default.htm* liefert den Einsprung für den Anwender (siehe oben).
- Die Datei *setup.exe* startet die Installation über die Datei *default.htm*.
- *ResClient.application* ist eine XML-Datei, sie enthält das Deploymentmanifest, welches Setup und Update der Anwendung konfigurieren.
- Im Unterverzeichnis *\Application Files\ResClient_1_0_0_1* befindet sich das eigentliche Programm. Es besteht aus der *exe*-Datei *ResClient.exe.deploy*, der Klassenbibliothek *ResServer-.dll.deploy* und dem Application-Manifest *ResClient.exe.manifest*.

Es folgen mehrere Sicherheitswarnungen die Sie – zumindest für unser Beispiel – bedenkenlos ignorieren können (Schaltfläche *Ausführen* bzw. *Installieren* wählen):





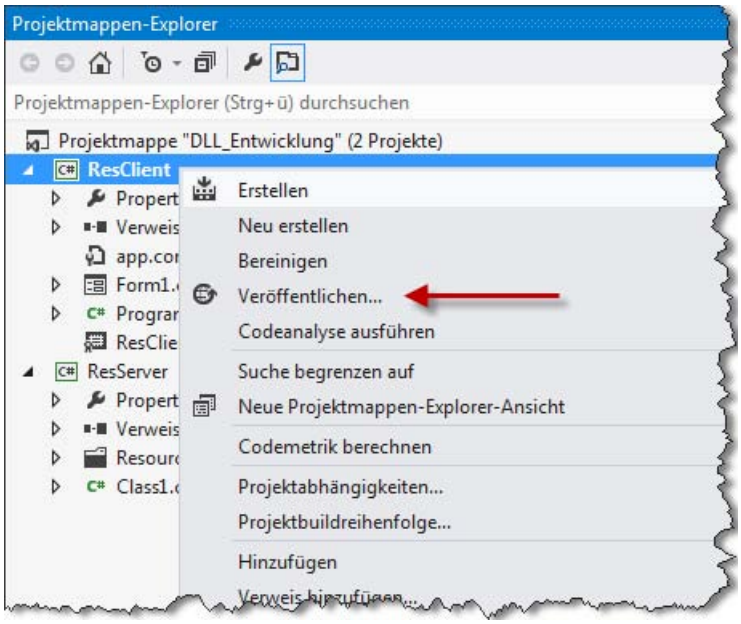
Nach dem Klick auf die *Installieren*-Schaltfläche und erfolgreichem Abschluss der Installation wird das Programm gestartet. Außerdem können Sie es – wie jedes andere Programm auch – über die *Start*-Schaltfläche des Windows Desktop aufrufen, selbst wenn keine Verbindung zum Netzwerk mehr besteht. Über *Systemeinstellungen|Software* lässt sich das Programm wieder deinstallieren, wobei Sie hier sogar die Option haben, die Vorgängerversion zu restaurieren.



27.1.10 Der Webpublishing-Assistent

Dieser Assistent soll vor allem dem Einsteiger die Arbeit erleichtern, da dieser die wichtigsten Deployment-Optionen nicht mehr unbedingt auf den Registerseiten *Sicherheit* und *Veröffentlichen* des Projekteigenschaften-Dialogs einstellen muss. Stattdessen erfolgt ein Frage-Antwort-Spiel.

Am einfachsten starten Sie den Assistenten über das Kontextmenü des *ResClient*-Projekts:



Nach Beantwortung einfacher Fragen ("Wird die Anwendung offline verfügbar sein?", "Wo möchten Sie die Anwendung veröffentlichen?") landet man auch hier bei *default.htm*, und es geht weiter wie oben bereits beschrieben.

27.1.11 Neue Versionen erstellen

Um sich in *ClickOnce* so richtig fit zu machen, sollten Sie bereits jetzt eine neue Version der Anwendung erstellen, zum Beispiel mit einer neuen Grafikressource im *ResServer*-Projekt oder einfach nur mit einer veränderten Beschriftung im *ResClient*. Nach erneutem Aufruf des Webpublishing-Assistenten werden Sie auf dem Webserver ein neues Verzeichnis *\ResClient_1_0_0_2* entdecken, in welchem die neue Programmversion abgelegt ist. Das Deployment-Manifest *ResClient.application* wurde so geändert, dass es jetzt auf das Application-Manifest der neuen Version verweist.

In Abhängigkeit von den vorgenommenen Einstellungen wird die Anwendung beim Aufruf sofort automatisch oder erst auf Nachfrage aktualisiert. Läuft etwas schief, so können Sie problemlos die Vorgängerversion zurückholen.

27.2 InstallShield

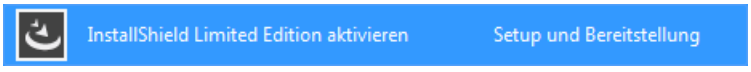
Nachdem wir uns im vorhergehenden Abschnitt intensiv mit dem ClickOnce-Deployment befasst haben, wollen wir uns nun mit dem von Visual Studio 2012 angebotenen InstallShield, dem Ersatz für die in Ungnade gefallenen klassischen Microsoft Setup-Projekte, beschäftigen. Mit Hilfe eines InstallShield-Projekts können Sie ein professionelles Setup-Programm erstellen.

Alle zu einem Setup-Projekt gehörenden Dateien sowie die Anweisungen zur Steuerung des Setups werden von InstallShield-Setup-Assistenten in eine *.msi*-Datei gepackt (das gilt auch im wörtlichen Sinne). Diese wiederum wird, zusammen mit dem InstallShield-Installationsprogramm, in eine *Setup.exe* verpackt. Spätestens diese Datei dürfte Ihnen als Windowsanwender bekannt vorkommen.

Doch bevor es soweit ist, müssen wir uns zunächst um die Installation und Registrierung von InstallShield kümmern, denn dies ist im Zusammenhang mit der Installation von Visual Studio noch nicht erfolgt.

27.2.1 Installation

Möchten Sie ein neues Setup-Projekt erstellen (*Datei|Neu|Projekt|Vorlagen|Andere Projekttypen|Setup und Bereitstellung*), wird Ihnen in Visual Studio 2012 lediglich die Möglichkeit geboten, die Anwendung herunterzuladen und zu aktivieren.



Klicken Sie auf obigen Eintrag, so landen Sie nach einigen Zwischenschritten auf der Flexera-Website, dem Anbieter von InstallShield.

Füllen Sie hier das Registrierungsformular aus (daran führt kein Weg vorbei, andernfalls erhalten Sie auch nicht den Key) und laden Sie dann die Installationsdatei herunter. Installieren Sie diese und starten Sie Visual Studio.

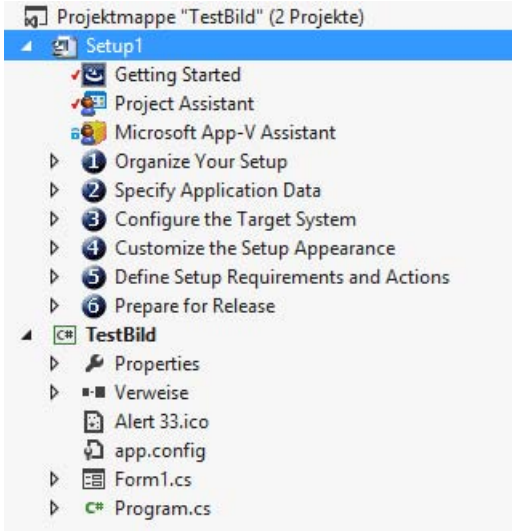
27.2.2 Aktivieren

Rufen Sie jetzt den Menüpunkt *Datei|Neu|Projekt|Vorlagen|Andere Projekttypen|InstallShield Limited Edition* auf, können Sie InstallShield registrieren (den Key erhalten Sie per E-Mail):

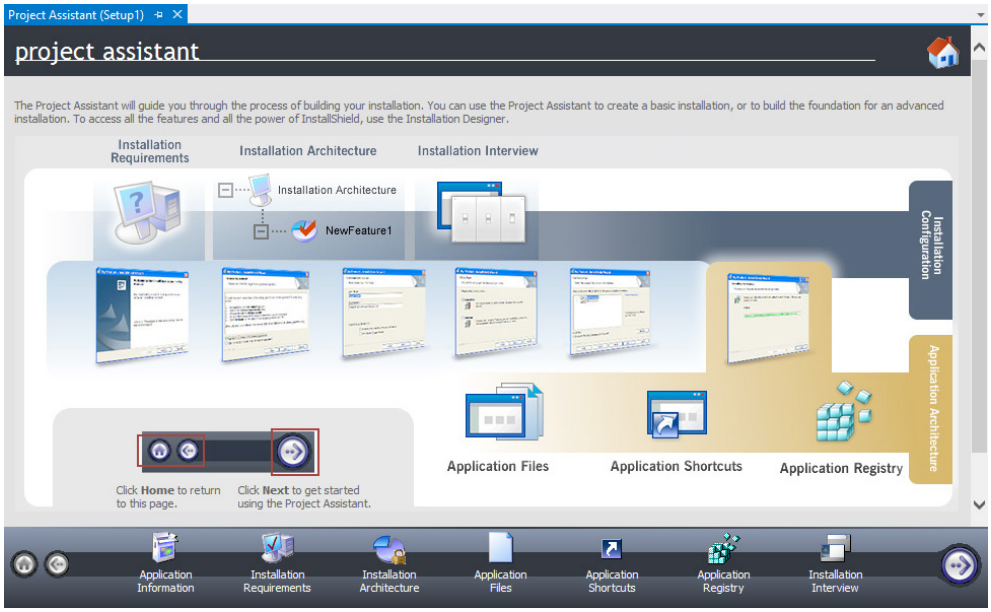
27.2.3 Ein neues Setup-Projekt

Öffnen Sie zunächst das C#-Projekt für das ein Setup erstellt werden soll. Starten Sie danach den Menüpunkt *Datei|Neu|Projekt...* und wählen Sie unter *Andere Projekttypen|InstallShield Limited Edition* die Vorlage *nstallShield Limited Edition Project* (zur Projektmappe hinzufügen), wenn Sie eine Windows-Anwendung vertreiben möchten.

Nachfolgend finden Sie im Projektmappen-Explorer ein neues Projekt vor:

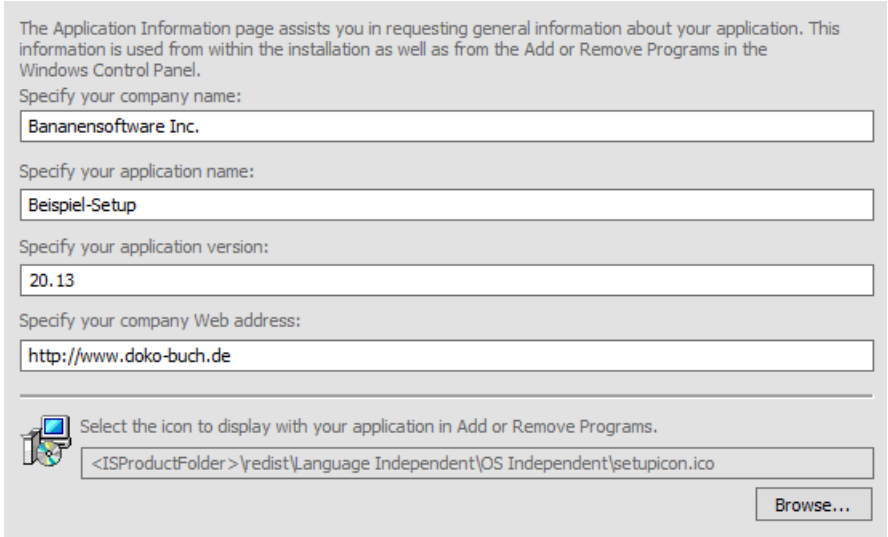


Gleichzeitig öffnet sich im Bearbeitungsbereich der Projekt-Assistent (ein recht buntes und wenig übersichtliches Fenster):



Klicken Sie auf die Pfeil-Schaltfläche in der rechten unteren Ecke und spezifizieren Sie die grundlegenden Informationen zu Ihrer Anwendung:

Anwendungseinstellungen



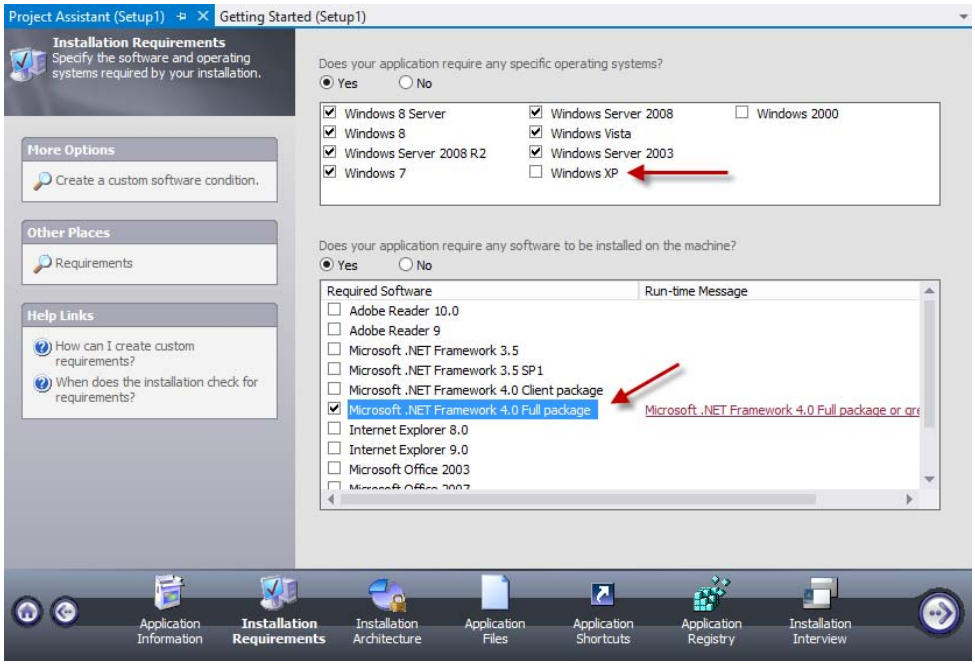
Die Bedeutung der einzelnen Eigenschaften dürfte recht schnell offenkundig sein.

HINWEIS: Weitere Einstellungen können Sie später direkt über den Projektmappen-Explorer vornehmen, hier werden nur die nötigsten Daten abgefragt.

Installationsbedingungen

So gern Sie auch Ihr Programm auf fremden PCs sehen, nicht in jedem Fall entspricht die Konfiguration des Endanwenders Ihren Festlegungen. So kann es sein, dass Sie Ihre Anwendung gezielt für Windows 7 entwickelt haben und Funktionen nutzen, die in Windows XP gar nicht enthalten sind (aktuelles Beispiel: das .NET-Framework 4.5). In diesem Fall sollte schon das Setup jeden Versuch verhindern, die Anwendung zu installieren.

Diese Aufgabenstellung übernimmt der Editor für die Installationsbedingungen. Hier legen Sie fest, welche Betriebssystem-Versionen unterstützt und welche Anwendungen installiert sein müssen:



HINWEIS: Dies sind nicht die Voraussetzungen, die installiert werden! Das zu installierende Framework etc. wählen Sie woanders aus.

Alternativ können Sie weitere Bedingen in Ihr Setup-Programm einbauen, dazu dient ein eigener Assistent, der Sie dabei unterstützt,

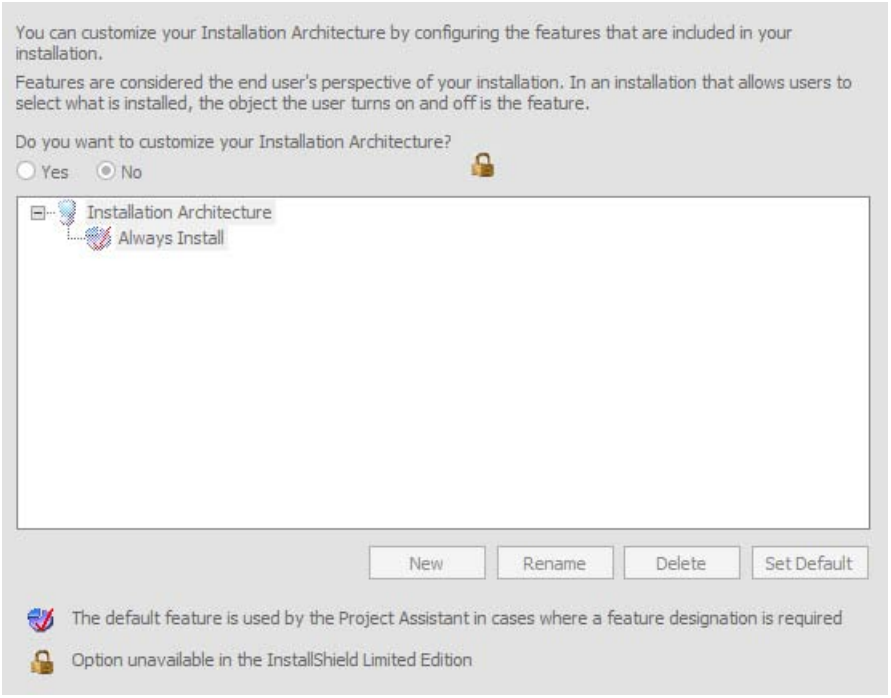
- Dateien,
- Verzeichnisse,

- Registrierdatenbank-Einträge und
- INI-Dateien

als Voraussetzung für eine Installation zu definieren.

Optionales Setup

Im nächsten Schritt müssen Sie es bei einer nicht konfigurierbaren Installation belassen, es sei denn, Sie erwerben eine Vollversion von InstallShield:

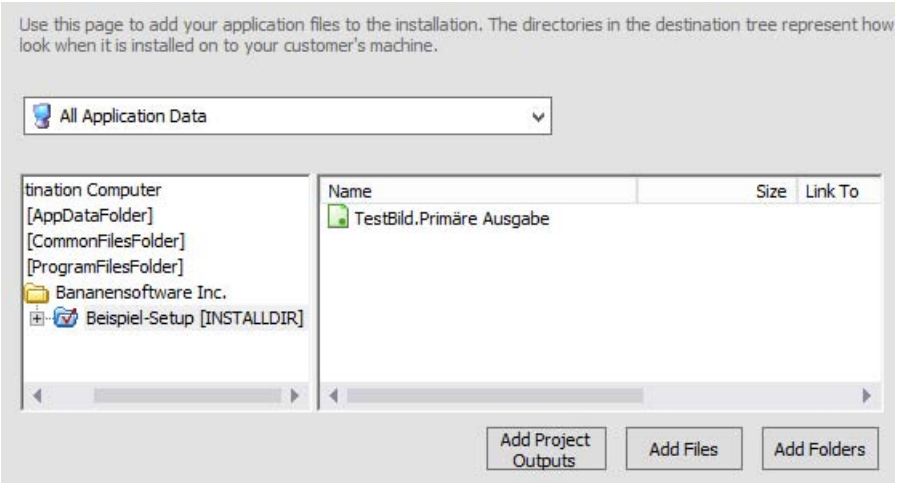


In diesem Fall können Sie verschiedene Optionsgruppen bilden, zwischen denen der Endanwender bei der Installation wählen kann (z.B. *Alles*, *Nur Anwendung* ...).

Anwendungsdateien

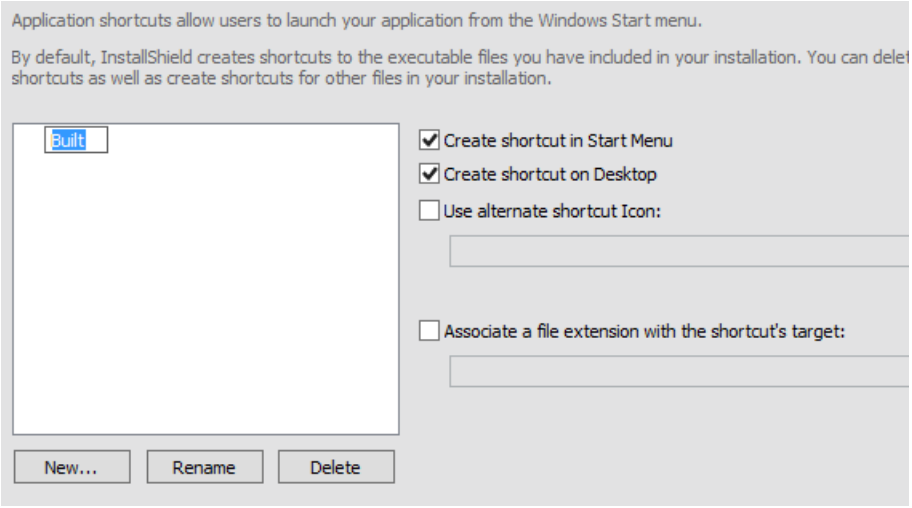
Wählen Sie im darauffolgenden Schritt, welche Dateien überhaupt auf dem Ziel-PC installiert werden sollen und in welchen Verzeichnissen dies zu erfolgen hat.

Wählen Sie dazu in der linken Liste zunächst das Zielverzeichnis (*[ProgrammFilesFolder]*), klicken Sie dann auf die Schaltfläche *Add Project Outputs* und wählen Sie zumindest die Option *Primäre Ausgabe*. Bestätigen Sie mit OK. Danach findet sich folgender Eintrag in der rechten Liste:



Startmenü und Verknüpfungen

Im nächsten Schritt des Assistenten können Sie eine Verknüpfung auf dem Desktop oder im Startmenü erstellen[1].

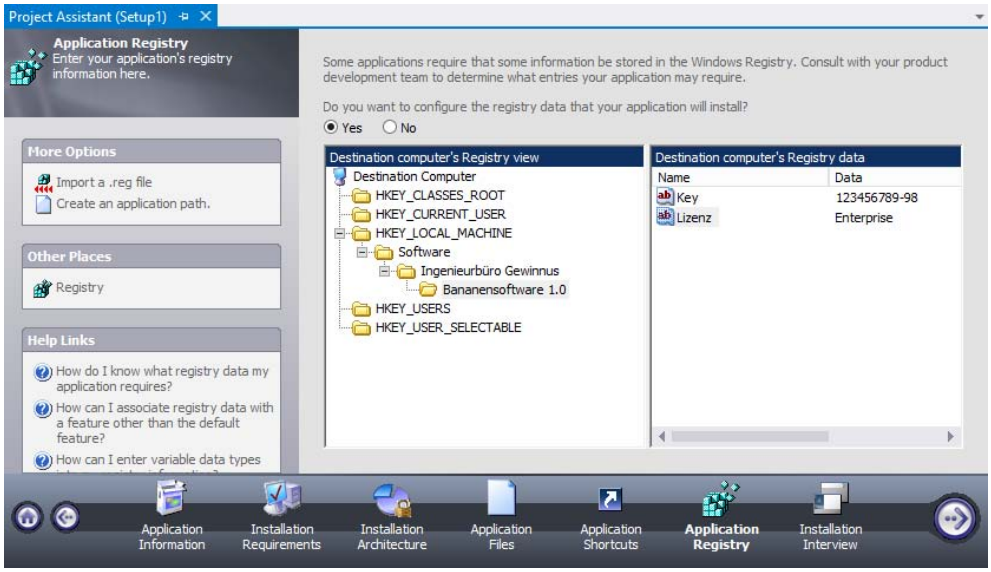


Optional können Sie hier auch noch ein anderes Symbol für die Verknüpfung festlegen.

Registrierungseinträge

Auch das Erzeugen von Registry-Einträgen ist mit den Setup-Projekten kein Problem. Öffnen Sie den entsprechenden Schritt des Projekt-Assistenten. Es erwartet Sie ein reduziertes Abbild der Registrierdatenbank mit den für eine Installation wichtigen Baumzweigen:

[1] Wenn Sie noch ein Startmenü haben – siehe Windows 8!

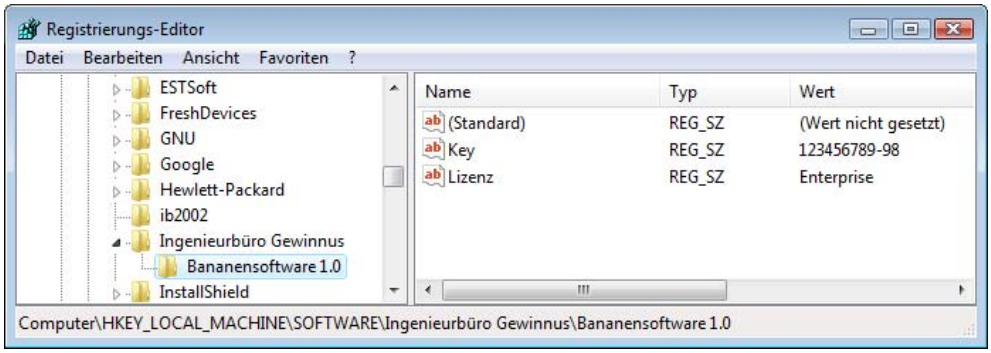


Legen Sie Einträge, die alle Nutzer betreffen (Installationspfade etc.), im Baumzweig HKEY_LOCAL_MACHINE ab, nutzerspezifische Daten (letzte geöffnete Datei etc.) im Baumzweig HKEY_CURRENT_USER. Für das Format der Registry-Einträge hat sich die folgende Notation eingebürgert:

```
…\<Software>\<Herstellername>\<Anwendungsname>\<Version>
```

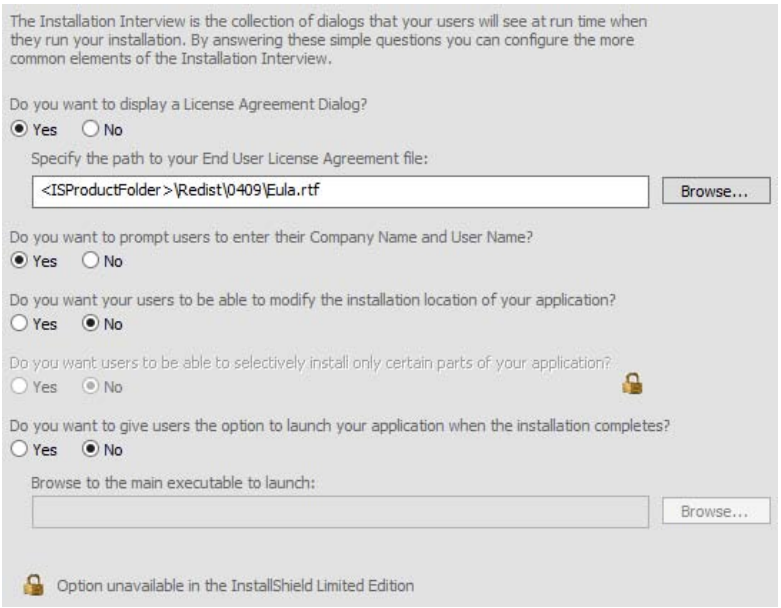
Im obigen Beispiel haben wir auch gleich noch zwei Values (*Key* und *Lizenz*) erzeugt.

Nach dem Generieren des Setup-Programms und der Installation finden sich die vordefinierten Registry-Einträge später auch an den gewünschten Stellen wieder:

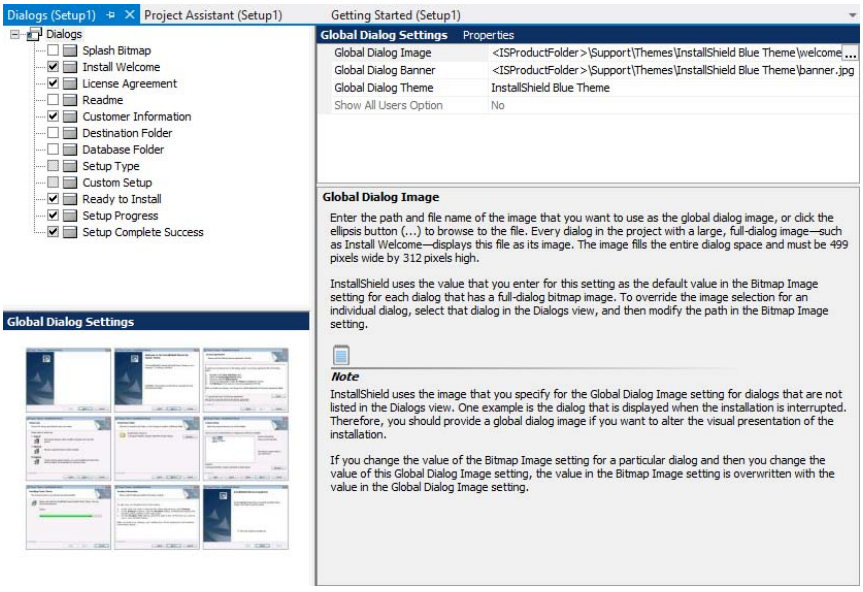


Setup-Dialoge/Erscheinungsbild

Welche Dialoge vom Setup benötigt werden, welche Lizenzbedingungen angezeigt werden etc., all dies lässt sich im letzten Schritt des Assistenten festlegen:



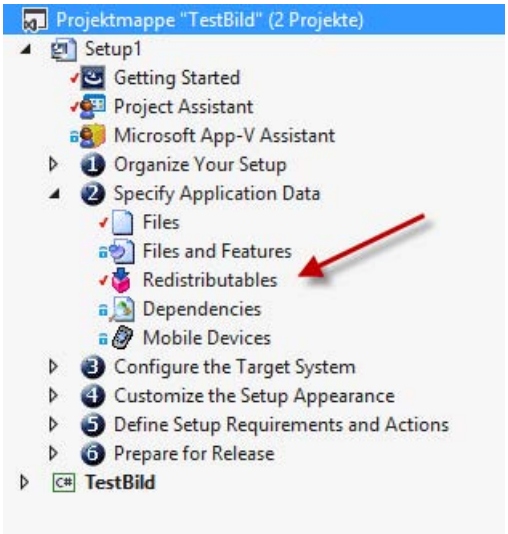
Von hier aus können Sie auch zur Dialogauswahl springen, eine Möglichkeit, die Ihnen auch per Projektmappen-Explorer offen steht.



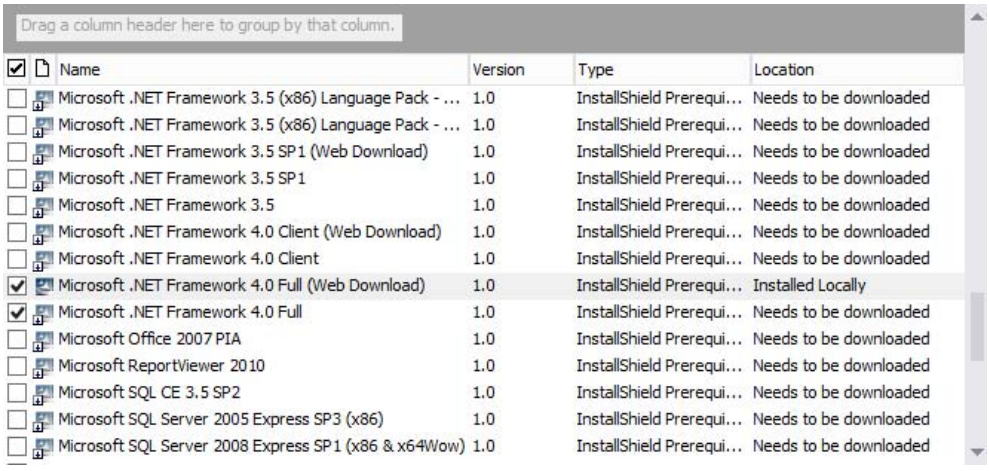
Installationsdateien

Eigentlich könnte jetzt schon der erste Test folgen, aber es fehlt noch etwas: Bekanntermaßen benötigen Ihre .NET-Programme auch das .NET-Framework und Sie können nicht davon ausgehen, dass dieses schon beim Endanwender in der gewünschten Version installiert ist.

Öffnen Sie also im Projektmappen-Explorer den entsprechenden Eintrag:



Wählen Sie in der nachfolgend angezeigten Liste das erforderliche Framework aus[1]:

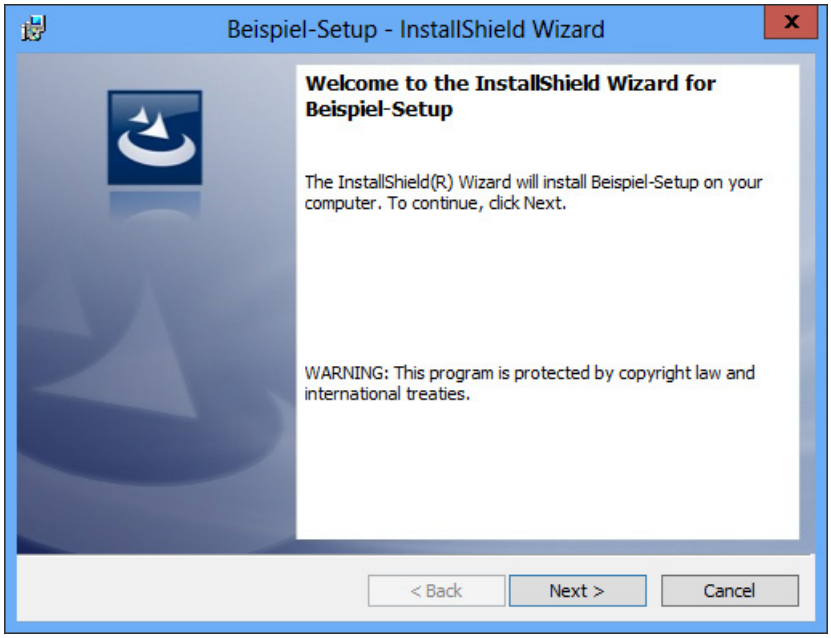


[1] In unserem Fall fehlte noch die Version 4.5, das aber dürfte sich bei Erscheinen des Buches geändert haben.

27.2.4 Finaler Test

Nach dem Erstellen des Setup-Projekts befinden sich die Setup-Dateien im Release-Verzeichnis *...\Express\SingleImage\DiskImages\DISK1*.

Nach dem Aufruf von *Setup.exe* und einer eventuellen Installation des Frameworks finden Sie sich dann in Ihrem selbst erstellten Installationsprogramm (**.msi*) wieder:



Die weiteren Schritte hängen dann davon ab, welche Dialoge Sie in das Setup mit eingeschlossen haben.

HINWEIS: Für ein wirklich sinnvolles Setup-Programm werden Sie nicht um den Erwerb der Vollversion von InstallShield herumkommen – in der **Limited** Edition sind einfach zu viele Optionen gesperrt.

Kapitel 28

Weitere Techniken

In diesem Kapitel stellen wir Ihnen in loser Folge einige interessante Themen vor, die sich nicht ohne weiteres in andere Kapitel einordnen ließen:

- Arbeit mit der Zwischenablage,
- Verwendung der Registrierdatenbank,
- Prinzip des .NET-Reflection-Mechanismus,
- Serielle Schnittstelle (RS232) .

28.1 Zugriff auf die Zwischenablage

Die Zwischenablage dient der Übertragung von Datenobjekten zwischen verschiedenen Anwendungen, entweder auf Grund von Befehlseingaben des Benutzers (*Einfügen*, *Kopieren*) oder programmgesteuert. Für diesen Zweck werden globale Speicherblöcke verwendet, auf die jede angemeldete Applikation zugreifen kann.

28.1.1 Das Clipboard-Objekt

Die Verbindung zur Windows-Zwischenablage hält unter .NET das *Clipboard*-Objekt. Neben den universellen Methoden

- *SetDataObject* und
- *GetDataObject*

bieten sich jede Menge typspezifischer Methoden an. Zu jedem der Datentypen *Data*, *Text*, *Audio*, *Image*, *FileDropList* gibt es eine *Contains*-, eine *Get*- und eine *Set*-Methode, z.B. *ContainsText()*, *GetText()* und *SetText()*.

Eine *Contains*- Methode (*true/false*) überprüft, ob in der Zwischenablage eine Information im gewünschten Format vorliegt. Die *Get*- und *Set*-Methoden übernehmen das Kopieren bzw. Einfügen.

Kopieren

Möchten Sie einen Text oder den Inhalt eines Textfeldes in die Zwischenablage kopieren, genügt der Aufruf von *SetText*, wie es das folgende Beispiel zeigt.

BEISPIEL 28.1: Kopieren von Text

C#
```
private void button1_Click(object sender, EventArgs e)
{
   if (textBox1.SelectedText != String.Empty)
      Clipboard.SetText(textBox1.SelectedText);
   else
      MessageBox.Show("Kein Text selektiert!");
}
```

HINWEIS: Es muss ein Objekt (Text, Grafik etc.) vorhanden sein, wenn Sie versuchen, die Daten zu kopieren. Andernfalls werden Sie mit einem Laufzeitfehler beglückt.

Einfügen

Das Einfügen von Daten aus der Zwischenablage gestaltet sich naturgemäß etwas schwieriger, da man nicht unbedingt weiß, was in der Zwischenablage enthalten ist.

Prüfen Sie zunächst, ob der gewünschte Datentyp vorhanden ist. Dazu müssen Sie

- die Methode *GetDataObject* aufrufen, diese gibt wiederum ein *IDataObjekt* zurück, das über seine Methode *GetDataPresent* auf ein bestimmtes Datenformat prüft,
- oder Sie verwenden eine der *Contains...*-Methoden, um gleich auf einen speziellen Datentyp (Text, Grafik) zu prüfen.

Der eigentlich Abruf der Daten kann dann mit den jeweiligen *Get...*-Methoden erfolgen.

BEISPIEL 28.2: Einfügen von Text

C#
```
private void button2_Click(object sender, EventArgs e)
{
            if (Clipboard.ContainsText())
                 textBox2.Text = Clipboard.GetText();
            else
                 MessageBox.Show("Keine geeigneten Daten im Clipboard!");
}
```

BEISPIEL 28.3: Eine Bitmap, von irgendwoher (z.B. aus einem Word-Dokument) in die Zwischenablage kopiert, wird in einer *PictureBox* angezeigt.

C#
```
if (Clipboard.ContainsImage())
{
   System.Drawing.Image pic = Clipboard.GetImage();
   pictureBox1.Image = pic;
```

BEISPIEL 28.3: Eine Bitmap, von irgendwoher (z.B. aus einem Word-Dokument) in die Zwischenablage kopiert, wird in einer *PictureBox* angezeigt.

```
}
else
   MessageBox.Show("Keine Bitmap im Clipboard!");
```

28.1.2 Zwischenablage-Funktionen für Textboxen

Im Zusammenhang mit der Zwischenablage finden sich unter dem Menüpunkt *Bearbeiten* fast immer die folgenden Befehle:

- Kopieren (*Strg+C*)
- Ausschneiden (*Strg+X*)
- Einfügen (*Strg+V*)

Wie trivial diese umzusetzen sind, entnehmen Sie den folgenden Beispielen:

BEISPIEL 28.4: Kopieren

C#
```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Copy();
}
```

BEISPIEL 28.5: Ausschneiden

C#
```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Cut();
}
```

BEISPIEL 28.6: Einfügen

C#
```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Paste();
}
```

28.2 Arbeiten mit der Registry

Auch wenn .NET mittlerweile andere Alternativen zur Registrierdatenbank zu bieten hat, dürfte diese noch lange im Gebrauch bleiben. Sei es, dass Sie bestehende Einträge auslesen müssen oder dass Sie einfach beim Umstellen vorhandener Programme nicht alles Know-how über Bord werfen wollen.

.NET bietet Ihnen mit den Klassen *Registry* und *RegistryKey* eine objektorientierte Schnittstelle, sodass Sie auf die Verwendung von API-Funktionen verzichten können.

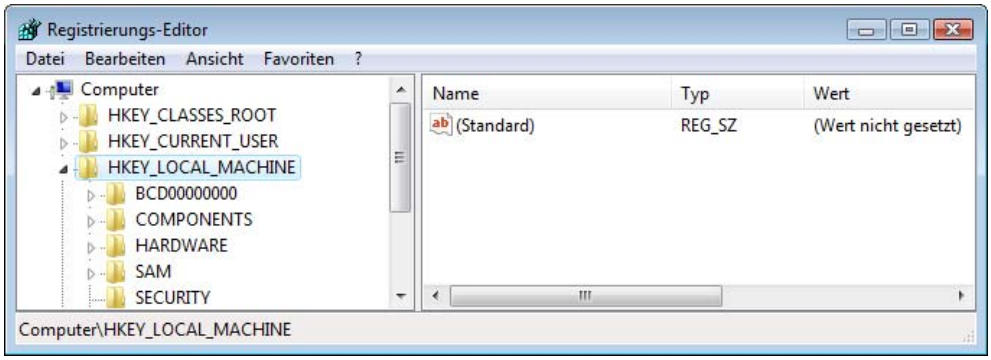
HINWEIS: Vergessen Sie nicht, den Namespace *Microsoft.Win32* in Ihr Programm aufzunehmen, wenn Sie die beiden Klassen nutzen wollen.

28.2.1 Allgemeines

Windows speichert alle Informationen über Hard- und Software in einer Datenbank mit Baumstruktur. Damit sollte endlich Ordnung in das INI-Datei-Chaos gebracht werden. Im Unterschied zu den INI-Dateien ist zum Einsehen der Datenbank ein Texteditor allerdings nicht geeignet, Sie müssen den Registrierungseditor verwenden, den Sie im Systemverzeichnis finden.

HINWEIS: Um den Windows-Registrierungseditor auszuführen, starten Sie das Programm *regedit.exe*!

Die Informationen sind in einer Baumstruktur gespeichert. Die einzelnen Einträge auf der linken Seite werden als "Schlüssel" bezeichnet, auf der rechten Seite werden gegebenenfalls die zugeordneten Werte angezeigt:



HINWEIS: Änderungen in der Registry sollten Sie nur vornehmen, wenn Sie sich über die Konsequenzen absolut im Klaren sind. Normalerweise werden die Einträge von den Programmen selbst verwaltet, Änderungen durch den Anwender sind nicht die Regel.

Die Registrierungsdatenbank gliedert sich in folgende vier wesentliche Teile:

Element	Beschreibung
HKEY_LOCAL_MACHINE	... enthält die Konfigurationsinformationen für den Computer (alle User).
HKEY_USERS	... die Root für alle Benutzerprofile.

Element	Beschreibung
HKEY_CURRENT_USER	... ist die Root der Konfigurationsinformationen für den momentan angemeldeten Benutzer (z.B. Programmgruppen, Bildschirmfarben ...). HKEY_CURRENT_USER ist ein Teilschlüssel von HKEY_USERS.
HKEY_CLASSES_ROOT	... ist ein Teilschlüssel von HKEY_LOCAL_MACHINE\SOFTWARE. Hier sind die Dateiverknüpfungen und OLE-Objekte gespeichert.

Bevor Sie einfach drauflos speichern, sollten Sie sich darüber im Klaren sein, **wo was** gespeichert wird. Grundsätzlich sollten Sie nur Informationen speichern, die auch in eine Registrierdatei gehören, umfangreiche Daten sollten auch weiterhin in einer eigenen Datei untergebracht sein.

Registrierungsdaten unterteilen Sie zweckmäßigerweise in

- User-spezifische Daten und
- Computer-spezifische Daten.

Computerdaten (z.B. Hardware-Infos) speichern Sie unter KEY_LOCAL_MACHINE nach folgendem Muster:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Firmenname\Produkt\Version\...
```

Daten, die den einzelnen User betreffen (z.B. Farbeinstellungen, Fenstergrößen etc.), speichern Sie unter HKEY_CURRENT_USER wie folgt:

```
HKEY_CURRENT_USER\SOFTWARE\Firmenname\Produkt\Version\...
```

Der Schlüssel SOFTWARE existiert bereits, *Firmenname* stellt eine eindeutige Firmenbezeichnung dar (z.B. *Microsoft*). Danach folgt die Produktbezeichnung (z.B. *Access*) und die Versionsnummer (z.B. *1.23*).

Auf diese Weise lassen sich alle Produkte einer Firma sinnvoll zusammenfassen.

BEISPIEL 28.7: Die nachfolgende Zeile steht für einen korrekten Eintrag.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0
```

Wichtig ist, dass in einer Baumebene alle Schlüssel eindeutig sein müssen, in einem Verzeichnis können Sie ja auch nicht zwei gleichnamige Dateien speichern.

Der Untereintrag eines Schlüssels kann hingegen den gleichen Namen wie der übergeordnete Schlüssel haben.

28.2.2 Registry-Unterstützung in .NET

Für alle Hauptzweige des Registry-Baums bietet die *Registry*-Klasse eigene Eigenschaften, die ein *RegistryKey*-Objekt zurückgeben:

Eigenschaft	Für den Zugriff auf ...
ClassesRoot	HKEY_CLASSES_ROOT
CurrentConfig	HKEY_CURRENT_CONFIG
CurrentUser	HKEY_CURRENT_USER
DynData	HKEY_DYN_DATA
LocalMachine	HKEY_LOCAL_MACHINE
PerformanceData	HKEY_PERFORMANCE_DATA
Users	HKEY_USERS

BEISPIEL 28.8: Zugriff auf den Baumzweig HKEY_CURRENT_USER

C#
```
using Microsoft.Win32;
...
    RegistryKey reg;
    reg = Registry.CurrentUser;
...
```

Mit diesem Objekt können Sie dann schon etwas mehr anfangen, wie die folgenden Tabellen zeigen:

Eigenschaft	Beschreibung
Name	... der Name des jeweiligen Schlüssels.
SubKeyCount	... die Anzahl der direkt untergeordneten Schlüssel.
ValueCount	... die Anzahl der Werte in diesem Schlüssel.

Methode	Beschreibung
Close	... Schließen des Keys.
CreateSubKey	... erzeugt einen neuen untergeordneten Schlüssel falls nötig, andernfalls wird der Schlüssel geöffnet.
DeleteSubKey	... löscht einen untergeordneten Schlüssel.
DeleteSubKeyTree	... löscht einen untergeordneten Schlüssel mit allen weiteren Schlüsseln (Vorsicht!).
DeleteValue	... löscht einen Wert.
GetSubKeyNames	... eine Stringliste aller Untereinträge (Schlüssel).
GetValue	... gibt einen Wert zurück.
GetValueNames	... eine Stringliste aller enthaltenen Werte.

Methode	Beschreibung
OpenSubKey	... öffnet einen Untereintrag.
SetValue	... setzt einen Wert.

HINWEIS: Ein Praxisbeispiel zur Registry finden Sie unter 28.5.1.

28.3 .NET-Reflection

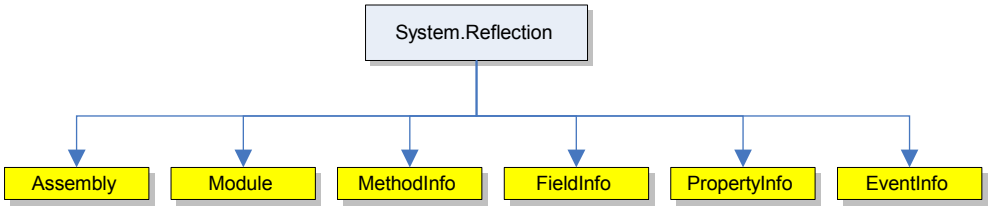
Bei manch hyperaktivem Datenschützer schrillen jetzt wahrscheinlich die Alarmglocken, denn in .NET ist ein perfides "Spitzelsystem" integriert, das die Möglichkeit bietet, jederzeit Informationen zu allen Metadaten einer Anwendung/Assembly zu gewinnen. Man nennt diesen Vorgang auch Reflexion. Die dazu erforderlichen Klassen finden Sie im *System.Reflection*-Namensraum.

HINWEIS: Reflection funktioniert selbst dann, wenn die Typen in externen Assemblierungen gespeichert sind und Sie den zugrunde liegenden Quellcode nicht kennen.

28.3.1 Übersicht

.NET-Reflections ermöglichen neben dem Abrufen von Typ-Informationen zur Laufzeit auch den Aufruf von Methoden per *Invoke*. Damit haben Sie alle Werkzeuge zur Hand, um zum Beispiel AddIns oder spezielle Filter zu programmieren, die jederzeit austauschbar sind (siehe Abschnitt 28.3.4, Dynamisches Laden von Assemblies).

Die wichtigsten Klassen auf einen Blick:



28.3.2 Assembly laden

Bevor Sie sich in die Untiefen der *Reflection*-Klassen stürzen, müssen Sie entscheiden, ob es sich um die aktuell geladene Assembly oder eine externe ungeladene Assembly handelt. Im letzteren Fall müssen Sie zunächst die gewünschte Assembly mit dem Methodenaufruf *Assembly.LoadFrom* laden. Der Rückgabewert ist eine Instanz der *Assembly*-Klasse.

BEISPIEL 28.9: Laden einer Assembly

C#
```
...
using System.Reflection;
```

BEISPIEL 28.9: Laden einer Assembly

```
...
private Assembly myass;
...
myass = Assembly.LoadFrom("xxxxx");
...
```

Geht es um die aktuell ausgeführte Assembly, können Sie mit *GetExecutingAssembly* die Instanz abrufen:

```
myass = Assembly.GetExecutingAssembly();
```

28.3.3 Mittels GetType und Type Informationen sammeln

Nachdem wir eine Instanz der *Assembly*-Klasse gebildet haben, können wir uns den "inneren" Werten zuwenden. Erster Anlaufpunkt ist die Methode *GetTypes*, die ein Array aller enthaltenen Klassen zurück gibt.

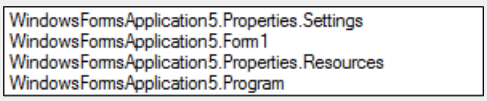
BEISPIEL 28.10: Auflisten aller Typen der aktuellen Assembly

C#

```
private void button1_Click(object sender, EventArgs e)
{
    Assembly myass = Assembly.GetExecutingAssembly();
    Type[] mytyp = myass.GetTypes();
    foreach (Type t in mytyp)
        listBox1.Items.Add(t.FullName);
}
```

Ergebnis

Die Ausgabe:



Alternativ können Sie mit *GetType* auch gezielt einen bestimmten Typ abrufen.

BEISPIEL 28.11: Abrufen der Klasse *Form1* aus der aktuellen Assembly

C#

```
myass = Assembly.GetExecutingAssembly();
Type tp = myass.GetType("ReflectionBsp.Form1");
```

Die *Type*-Klasse hilft Ihnen mit den folgenden, selbsterklärenden Eigenschaften weiter:

- *Name, FullName*
- *Namespace*
- *IsClass*

- *IsInterface*
- *IsAbstract*
- *IsCOMObject*
- *IsEnum*
- *IsSealed*
- *IsPublic*

Einige wichtige Methoden:

- *GetMembers*
- *GetFields*
- *GetProperties*
- *GetMethods*

Alle Methoden liefern ein Array des jeweils abgerufenen Typs zurück.

BEISPIEL 28.12: Das aktuelle Formular soll näher untersucht werden.

C#

```
myass = Assembly.GetExecutingAssembly();
Type tp = myass.GetType("ReflectionBsp.Form1");  // oder tp = this.GetType();
listBox1.Items.Clear();
```

Name in die *ListBox* eintragen:

```
listBox1.Items.Add("      Typ = " + tp.Name.ToString());
```

Falls es sich um eine Klasse handelt:

```
if (tp.IsClass)
{
```

Alle Eigenschaften:

```
    listBox1.Items.Add("------- Eigenschaften ------");
    foreach (PropertyInfo p in tp.GetProperties())
        listBox1.Items.Add("  - " + p.Name.ToString());
```

Alle Methoden:

```
    listBox1.Items.Add("-------- Methoden --------");
    foreach (MethodInfo m in tp.GetMethods())
    {
        if (m.IsPublic & !m.IsSpecialName)
            listBox1.Items.Add("  - " + m.Name.ToString());
        else
            listBox1.Items.Add("  - " + m.Name.ToString() +
                  "    (not Public or SpecialName)");
    }
```

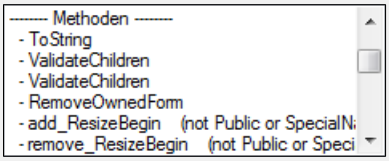
BEISPIEL 28.12: Das aktuelle Formular soll näher untersucht werden.

Alle Felder:

```
            listBox1.Items.Add("-------- Felder --------");
            foreach (FieldInfo f in tp.GetFields())
                listBox1.Items.Add("  - " + f.Name.ToString());
        }
```

Ergebnis

Die Ausgabe (Ausschnitt):



28.3.4 Dynamisches Laden von Assemblies

Nicht immer steht schon zur Entwurfszeit fest, welche Klasse/Funktion für eine bestimmte Aufgabe eingesetzt werden soll. Aus der guten alten Win32-Welt kennen Sie bestimmt noch das dynamische Laden von DLLs mittels *LoadLibrary*. Mit der Funktion *GetProcAddress* konnten Sie den Zeiger auf eine enthaltene Funktion ermitteln und dann diese Funktion dynamisch aufrufen.

Das .NET-Framework stellt Ihnen die gleiche, jedoch dank Reflection wesentlich leistungsfähigere Funktionalität zur Verfügung. Von Interesse sind in diesem Zusammenhang

- die *Assembly*-Klasse (zum Laden der gewünschten Assembly),
- die *Activator*-Klasse (zum Erzeugen der eigentlichen Instanz) und last but not least
- die *Invoke*-Methode (zum Aufrufen der gewünschten Methode).

Ein kleines Beispielprogramm zeigt die Vorgehensweise.

Beispiel

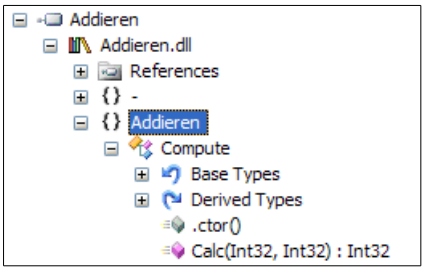
Eine Windows Forms-Anwendung soll zwecks Ermittlung der Summe bzw. der Differenz eine von zwei Assemblies (*Addieren.dll*, *Subtrahieren.dll*) dynamisch laden, eine Instanz der jeweils enthaltenen Klasse erzeugen und das Ergebnis mittels Methodenaufruf berechnen.

Zunächst erstellen wir die beiden o.g. Assemblies (wählen Sie den Projekttyp *Klassenbibiothek*). Fügen Sie jeweils eine Klasse hinzu:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Addieren
{
```

```
    public class Compute
    {
        public int Calc(int a, int b)
        {  return a + b;  }
    }
}
```

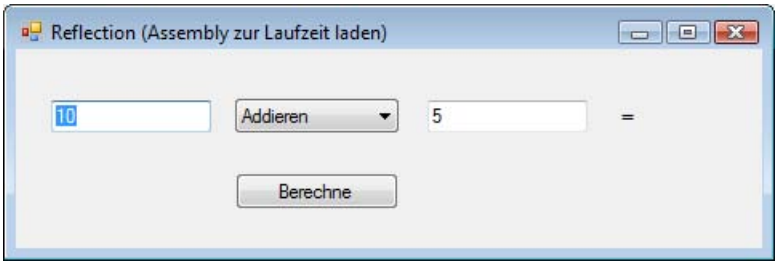
Das Ergebnis im .NET-Reflector:



HINWEIS: Die Klasse *Subtrahieren* ist ähnlich einfach aufgebaut, Sie finden die Daten in den Buchbeispielen.

Kompilieren Sie die beiden Assemblies, und kopieren Sie später die DLLs in das Projektverzeichnis der eigentlichen Anwendung.

Für unser aufrufendes Windows Forms-Programm entwerfen Sie bitte die folgende einfache Oberfläche (Abbildung). Mit der *ComboBox* wählen wir zur Laufzeit die Assembly aus, mit dem *Button* starten wir die Berechnung.



Der Code:

```
...
using System.Windows.Forms;
using System.Reflection;
using System.IO;

namespace ReflectionRechner
{
    public partial class Form1 : Form
```

Mit dem Laden des Formulars informieren wir uns zunächst, welche Assemblies zur Verfügung stehen und tragen deren Namen in die *ComboBox* ein. So können später jederzeit weitere Bibliotheken hinzugefügt werden.

```
private void Form1_Load(object sender, EventArgs e)
{
    DirectoryInfo myDir;
    FileInfo[] myFiles;
    myDir = new DirectoryInfo(Application.StartupPath);
    myFiles = myDir.GetFiles("*.dll");
    foreach (FileInfo f in myFiles)
        comboBox1.Items.Add(f.Name.Substring(0, f.Name.Length - 4));
    comboBox1.SelectedIndex = 0;
}
```

Mit dem Klick auf den *Button* wird es ernst:

```
private void button1_Click(object sender, EventArgs e)
{
```

Laden des Typs aus der gewünschten Assembly:

```
    Type typ = Assembly.LoadFrom(comboBox1.Text + ".dll").GetType(comboBox1.Text +
                                 ".Compute");
```

Instanz erstellen:

```
    object obj = Activator.CreateInstance(typ);
```

Methode abrufen (hier sehen Sie die Vorteile von Reflection):

```
    MethodInfo meth = typ.GetMethod("Calc");
```

Die Eingabewerte in *Integer*-Werte umwandeln:

```
    int p1 = Convert.ToInt32(textBox1.Text);
    int p2 = Convert.ToInt32(textBox2.Text);
```

Die Methode *Calc* aufrufen und den Rückgabewert auswerten:

```
    int p3 = (int) meth.Invoke(obj, new object[] { p1, p2 });
    label1.Text = "= " + p3.ToString();
}
```

Übergabewerte beim Methodenaufruf sind das betreffende Objekt und (in einem Objektarray) die Parameter/Übergabewerte.

HINWEIS: Anzahl, Anordnung und Datentyp im Objektarray müssen mit Anzahl, Anordnung und Typ der Parameter der aufzurufenden Methode übereinstimmen!

Jetzt endlich kann getestet werden: Starten Sie das Programm und klicken Sie auf die beiden *Button*s. Im *label1* sollte jetzt das richtige Ergebnis erscheinen.

Wie Sie sehen, ist es (fast) problemlos möglich, zur Laufzeit Objekte aus externen Assemblies zu erzeugen und deren Methoden auszuführen. Obiger Quellcode könnte zum Beispiel dahingehend geändert werden, dass auch die enthaltenen Namespaces identische Namen bekommen. In diesem Fall genügt schon die Änderung des Assembly-Namens beim Laden, um eine gänzlich andere Funktionalität bereitzustellen. So könnten Sie spezielle Im-/Exportfilter, Add-Ins etc. programmieren.

28.4 Das SerialPort-Control

Die aus dem Jahre 1962 stammende RS232-Schnittstelle ist zwar mittlerweile ein Veteran unter den seriellen Schnittstellen, aber aufgrund ihrer geringen Ansprüche an Hard- und Software nicht nur unter Bastlern sehr beliebt, sondern auch im Laboralltag technischer Lehreinrichtungen und bei industriellen Anwendern. Trotz USB verwenden deshalb viele moderne Geräte und Programme zum Messen, Steuern und Regeln auch heute noch diese Schnittstelle.

Die *SerialPort*-Klasse aus dem Namespace *System.IO.Ports* hat folgende Vorzüge:

- das umständliche Einbinden des "alten" *MSComm.ocx*-Controls entfällt,
- das *SerialPort*-Control ist ein "natives" .NET-Control,
- enthält neue ereignisgesteuerte Funktionalität und
- wird in der MSDN Library ausreichend dokumentiert.

28.4.1 Übersicht

Die folgenden Tabellen listen die wichtigsten Mitglieder dieser Klasse auf.

Wichtige Eigenschaften

Die Klasse *SerialPort* erlaubt den Direktzugriff auf die Eigenschaften des Treibers der seriellen Schnittstelle. Die folgende Tabelle zeigt die wichtigsten Properties.

Eigenschaft	Bedeutung (Standardwert in Klammern)
PortName	... ist der Name des seriellen Ports (Standardwert = "COM1")
BaudRate	... liest oder schreibt die serielle Baudrate (Bit pro Sekunde). Übliche Werte sind: 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 (Standardwert = 9600)
DataBits	Anzahl der Datenbits pro Byte (Standardwert = 8)
StopBits	... liest oder schreibt die Anzahl von Stoppbits pro Byte (Standardwert = *One*)
ReadTimeout	... liest oder schreibt die Anzahl von Millisekunden bis zu einem Timeout, wenn eine Leseoperation nicht abgeschlossen wird.
Parity	Paritätsprüfung (Standardwert = *None*)
DtrEnable	Data Terminal Ready -Signal (Standardwert = *False*)

HINWEIS: Mit wenigen Ausnahmen (*ReadTimeout*) haben alle öffentlichen Eigenschaften Standardwerte. Viele Geräte/Modems lassen sich z.B. mit 8N1 (8 Bits, keine Parity, 1 Stopbit) ansprechen.

Wichtige Methoden

Die folgende Tabelle zeigt nur einige wichtige Methoden (zahlreiche weitere Methoden der *SerialPort*-Klasse entnehmen Sie der Dokumentation):

Methode	Bedeutung
GetPortNames()	... liefert die Namen der im aktuellen System vorhandenen COM-Ports
ReadLine()	... liest Zeichen bis zu einem *NewLine*-Wert in den Input-Puffer. Falls bis zum Timeout kein *NewLine*-Wert gefunden wird, liefert die Methode *Null*.
WriteLine(string)	... schreibt einen String und einen *NewLine*-Wert in den Output-Puffer.
Read(byte[], start, anzahl)	... liest eine *anzahl* Bytes ab *start* in ein Byte-Array
Open()	... öffnet eine neue serielle Port-Verbindung
Close()	... schließt die Port-Verbindung

Wichtige Ereignisse

Im Zentrum der Programmierung steht das *DataReceived*-Ereignis (siehe folgende Tabelle):

Ereignis	Bedeutung
DataReceived	... wird ausgelöst, wenn im Empfangspuffer Daten vorhanden sind
ErrorReceived	... wird ausgelöst, wenn beim Datenempfang ein Fehler auftritt

28.4.2 Einführungsbeispiele

Wir wollen zunächst an einem simplen Beispiel zeigen, wie unkompliziert der Datenaustausch zwischen zwei seriellen Schnittstellen unter Verwendung der *SerialPort*-Klasse realisiert werden kann. Dazu schreiben wir zwei einfache Konsolenanwendungen: *SerialPortTalker* und *SerialPortListener*.

SerialPortTalker

Öffnen Sie eine neue Konsolenanwendung und tragen Sie folgenden Code ein.

```
using System.IO.Ports;

namespace SerialPortTalker

    class Program
    {
```

```
        static void Main(string[] args)
        {
```

Einer der insgesamt sieben überladenen Konstruktoren ermöglicht es, das *SerialPort*-Control mit den wichtigsten Standardeigenschaften zu instanziieren:

```
            SerialPort port = new SerialPort("COM1", 9600, Parity.None, 8, StopBits.One);
```

Port öffnen:

```
            port.Open();
```

Einen String senden:

```
            port.WriteLine("Hallo Serielle Schnittstelle! Wie geht es Dir?");
```

Port schließen:

```
            port.Close();
        }
    }
}
```

Wie Sie sehen, umfasst die Schreib-Kommunikation mit dem Port lediglich den aufeinanderfolgenden Aufruf der Methoden *Open*, *WriteLine* (mit Übergabe der zu übermittelnden Zeichenkette) und *Close*.

Kompilieren Sie das Programm (*F5*). Bevor aber ein sinnvoller Programmtest möglich ist, muss erst noch die zweite Konsolenanwendung, der *SerialPortListener*, fertig gestellt werden.

SerialPortListener

Tragen Sie in eine neue Konsolenanwendung den folgenden Code ein. Wie Sie sehen, erfolgt das Lesen bzw. der Empfang von asynchronen Daten, die über den seriellen Port hereinkommen, ereignisgesteuert.

```
using System;
using System.Windows.Forms;
using System.IO.Ports;

namespace SerialPortListener
{
    class Program
    {
```

SerialPort-Objekt mit Standardwerten erzeugen:

```
        private SerialPort port = new SerialPort("COM1", 9600, Parity.None, 8, StopBits.One);

        static void Main(string[] args)
        {
```

Das *Program* instanziieren (dabei wird der Konstruktor ausgeführt):

```
            new Program();
```

```
        }
```

Der Konstruktor:

```
        private Program()
        {
            Console.WriteLine("Empfangene Daten:");
```

Einen Eventhandler für das *DataReceived*-Ereignis anmelden:

```
            port.DataReceived += new SerialDataReceivedEventHandler(port_DataReceived);
```

Die Kommunikation kann beginnen:

```
            port.Open();
```

Eine Anwendungsschleife aufrufen, die das Beenden des Programms verhindert:

```
           System.Windows.Forms.Application.Run();
        }
```

Der Eventhandler für das *DataReceived*-Ereignis:

```
        private void port_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
```

Alle empfangenen Daten aus dem Empfangspuffer des Ports anzeigen:

```
            Console.Write(port.ReadExisting());
        }
    }
}
```

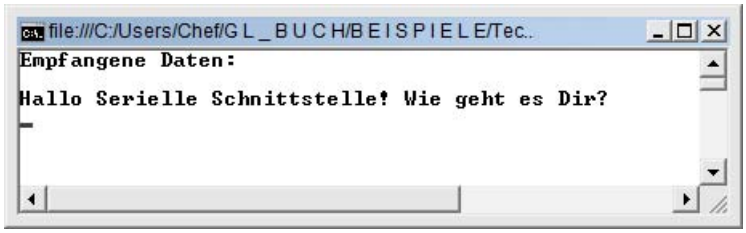
Test

Wollen Sie den Datenaustausch zwischen zwei realen PCs demonstrieren, so installieren Sie auf dem einen PC eine Programminstanz von *SerialPortTalker* und auf dem anderen eine Programminstanz von *SerialPortListener*.

Natürlich brauchen Sie auch ein Nullmodem-Kabel (hier sind beide Anschlüsse weiblich!), um die seriellen COM1-Ports beider PCs miteinander zu verbinden.

HINWEIS: Wer nicht über zwei PCs verfügt, verbindet einfach COM1 und COM2 seines Computers mit dem Nullmodem-Kabel und ändert im Programm *SerialPortListener* die *PortName*-Eigenschaft in "COM2".

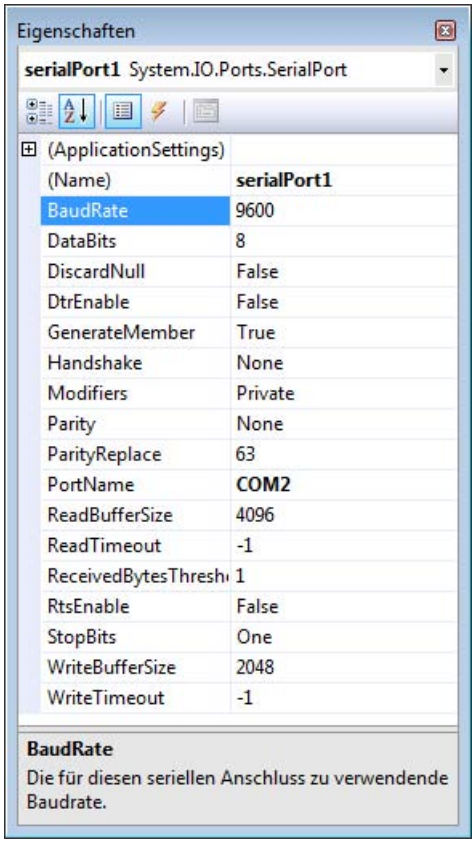
Starten Sie zunächst das Programm *SerialPortListener*, es öffnet sich das Konsolenfenster in Erwartung hereinkommender Daten. Anschließend starten Sie das Programm *SerialPortTalker*, nach einem kurzen Aufblitzen der Konsole hat es auch schon seine Mission erfüllt und ist beendet. Das Ergebnis sollte im *SerialPortListener* sichtbar sein.



28.4.3 Thread-Probleme bei Windows Forms-Anwendungen

Konsolenanwendungen sind seit langem nicht mehr zeitgemäß. Wir wollen deshalb zunächst den *SerialPortListener* durch eine Windows Applikation ersetzen.

Nichts ist einfacher als das, werden Sie denken und haben schnell auf das Startformular *Form1* ein *Label* für die Anzeige der Daten gesetzt, eine *SerialPort*-Komponente von der Toolbox abgezogen und deren Eigenschaften zugewiesen.

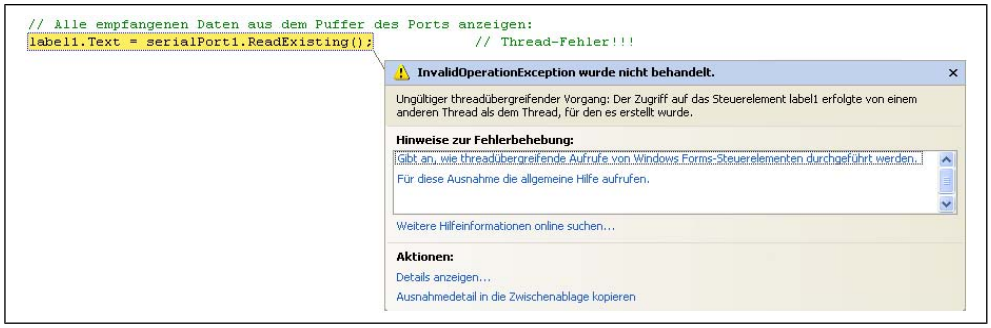


Auch der Quellcode ist schnell erstellt:

```
using System.Windows.Forms;
using System.IO.Ports;

namespace SerialPortRead_Win1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            serialPort1.Open();
        }

        private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
            // Alle empfangenen Daten aus dem Puffer des Ports anzeigen:
            label1.Text = serialPort1.ReadExisting();           // Thread-Fehler!!!
        }
    }
}
```

Zwar lässt sich der Code anstandslos kompilieren, das böse Erwachen kommt aber zur Laufzeit, wenn Daten hereinkommen (Sie starten dazu z.B. den *SerialPortTalker*) und der Inhalt des Labels gesetzt werden soll. Dabei wird folgende Exception geworfen:

Ungültiger threadübergreifender Vorgang: Der Zugriff auf das Steuerelement label1 *erfolgte von einem anderen Thread als dem Thread, für den es erstellt wurde.*



Wir haben es hier nicht etwa mit einem Bug der *SerialPort*-Komponente, sondern mit einem typischen Fehler zu tun, welcher ganz allgemein mit GUI-Änderungen in Forms bei multi-threaded Anwendungen zusammenhängt:

HINWEIS: GUI-Komponenten dürfen nur von dem Thread verändert werden, von dem sie erzeugt wurden!

Das Empfangen der seriellen Daten läuft aber in einem anderen Thread ab. Als Lösung des Problems bietet sich ein Methodenzeiger (Delegate) an, der über die *Invoke*-Methode der entsprechenden GUI-Komponente (hier *label1*) aufgerufen wird (siehe folgendes Listing):

```
using System.IO.Ports;
...
    public partial class Form1 : Form
    {
```

Delegate-Typ deklarieren:

```
        private delegate void showTxtDlg(string txt);
```

Delegate-Objekt/Methodenzeiger deklarieren:

```
        private showTxtDlg showTxtPtr;

        public Form1()
        {
            InitializeComponent();
```

Delegate-Objekt instanziieren:

```
            showTxtPtr = new showTxtDlg(showText);   // oder verkürzt: showTxtZ = showText;
            serialPort1.Open();
        }

        private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
```

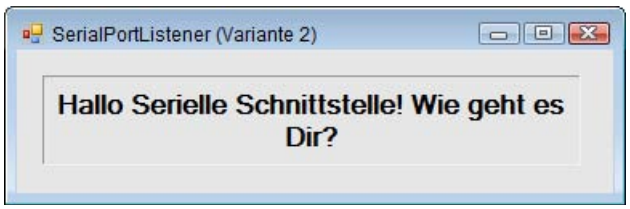
Der *Invoke*-Methode werden das Delegate-Objekt und die Anzeigedaten übergeben:

```
            label1.Invoke(showTxtPtr, serialPort1.ReadExisting());
        }
```

Die Anzeigeroutine, auf welche das Delegate-Objekt *showTxtPtr* verweist:

```
        private void showText(string txt)
        {
            label1.Text += txt;
        }
    }
}
```

Nun sollte auch das Abhören des Ports mittels Windows-Oberfläche funktionieren:



Reichlich umständlich, nicht wahr? Neben den drei erforderlichen Schritten (Delegate-Typ definieren; Delegate-Objekt deklarieren; Delegate-Objekt erzeugen), die zum gewünschten Methodenzeiger *showTxtPtr* führen, muss auch noch eine extra Anzeigeroutine *showText* geschrieben werden.

Der folgende Code zeigt eine deutlich einfachere und übersichtlichere Lösung für eine Thread-Umschaltung in verkürzter Notation, die sich u.a. auch *anonymer Methoden* bedient:

```
...
using System.IO.Ports;
...
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            serialPort1.Open();
        }

        private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
```

UI-Thread per *Invoke* aufrufen und Delegate übergeben:

```
            label1.Invoke(new EventHandler(delegate
            {
                label1.Text += serialPort1.ReadExisting();     // anonyme Methode!
            }
            ));
        }
    }
}
```

Dieser Code liefert das gleiche Ergebnis wie sein Vorgänger, allerdings hält sich der Schreibaufwand in Grenzen und ist kaum größer als bei der vergleichbaren Konsolenanwendung. Der Delegate verweist jetzt direkt auf die in { }-Klammern angehängte anonyme (d h. namenlose) Anzeigemethode.

28.4.4 Ein einfaches Terminalprogramm

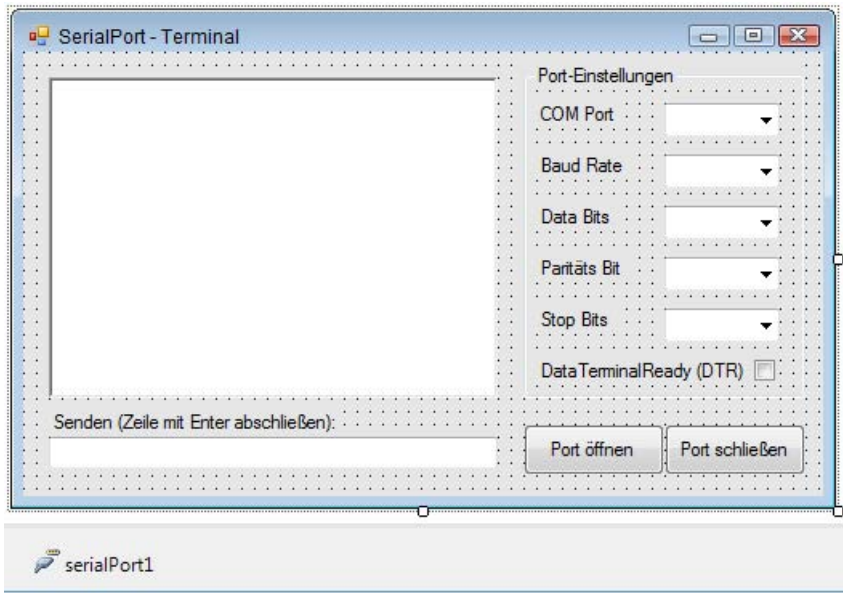
Nachdem wir nun die Threading-Klippen glücklich umschifft haben und die Anwendung der *SerialPort*-Klasse in Windows-Anwendungen vom Prinzip her einigermaßen klar ist, sollten wir uns an ein kleines Terminalprogramm wagen, welches die Funktionalitäten von *SerialPortListener* und *SerialPortTalker* kapselt und mit welchem man nicht nur chatten kann, sondern mit dem sich auch weitere aufschlussreiche Experimente durchführen lassen.

Bedienoberfläche

Eine *RichTextBox* (oben) und eine *TextBox*(unten) dienen zur Anzeige bzw. zur Eingabe der Daten. Außerdem wird eine *SerialPort*-Komponente benötigt, deren Eigenschaften mit fünf *ComboBox*en und einer *CheckBox* eingestellt werden, die in einer *GroupBox* angeordnet sind. Zwei *Button*s dienen zum Öffnen und Schließen der seriellen Schnittstelle. Die *Enabled*-Eigenschaft von *button2* setzen wir auf *False*.

Im Eigenschaftenfenster weisen wir der *Items*-Auflistung der *ComboBox*en die in der Tabelle gezeigten Werte zu:

Name	Beschriftung	Items-Auflistung
comboBox1	COM Port	siehe Quellcode!
comboBox2	Baud Rate	300, 2400, 9600, 19200, 38400, 57600, 115200
comboBox3	DataBits	7, 8, 9
comboBox4	Paritäts Bit	None, Even, Odd
comboBox5	Stop Bit	1, 2, 3



Quellcode

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.IO.Ports;
...
```

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
```

Dank *GetPortNames*-Methode ist es kein Problem, die vorhandenen Ports festzustellen und in *comboBox1* zur Auswahl anzubieten:

```
        foreach (string s in SerialPort.GetPortNames()) comboBox1.Items.Add(s);
```

Die Anfangswerte der *ComboBox*en einstellen:

```
        comboBox2.SelectedIndex = 1;   // 2400
        comboBox1.SelectedIndex = 0;   // COM1
        comboBox3.SelectedIndex = 1;   // 8 Datenbits
        comboBox4.SelectedIndex = 0;   // keine Parität
        comboBox5.SelectedIndex = 0;   // 1 Stopp-Bit
```

Benutzerschnittstelle anpassen:

```
        enableControls();
    }
```

Die *enableControls*-Methode aktiviert/deaktiviert die Komponenten der Benutzerschnittstelle je nach Öffnungsstatus des *SerialPort*:

```
    private void enableControls()
    {
        if (!serialPort1.IsOpen)          // Port ist geschlossen
        {
            groupBox1.Enabled = button1.Enabled = true;
            button2.Enabled = textBox1.Enabled = richTextBox1.Enabled = false;
        }
        else                              // Port ist geöffnet
        {
            groupBox1.Enabled = button1.Enabled = false;
            button2.Enabled = textBox1.Enabled = richTextBox1.Enabled = true;
            textBox1.Focus();
        }
    }
```

Port öffnen:

```
    private void button1_Click(object sender, EventArgs e)
    {
```

Einstellungen übernehmen:

```
        serialPort1.PortName = comboBox1.Text;
        serialPort1.BaudRate = int.Parse(comboBox2.Text);
        serialPort1.DataBits = int.Parse(comboBox3.Text);
        serialPort1.Parity = (Parity)Enum.Parse(typeof(Parity), comboBox4.Text);
        serialPort1.StopBits = (StopBits)Enum.Parse(typeof(StopBits), comboBox5.Text);
```

```
            serialPort1.DtrEnable = checkBox1.Checked;        // Data Terminal Ready - Signal
            serialPort1.Encoding = System.Text.Encoding.UTF8;       // Kodierung einstellen
            serialPort1.Open();
```

Eingabemaske anpassen:

```
            enableControls();
        }
```

Port schließen:

```
        private void button2_Click(object sender, EventArgs e)
        {
           if (serialPort1.IsOpen) serialPort1.Close();
           enableControls();
        }
```

Senden des Inhalts von *textBox1* nach Drücken der Enter-Taste:

```
        private void textBox1_KeyUp(object sender, KeyEventArgs e)
        {
            if ((e.KeyCode == Keys.Enter) && (textBox1.Text != String.Empty))
            {
                string data = textBox1.Text;
                serialPort1.WriteLine(data);
                richTextBox1.SelectionColor = Color.Green;
                richTextBox1.AppendText(data + Environment.NewLine);   // Anzeige
                richTextBox1.ScrollToCaret();              // Bildlauf
                textBox1.SelectAll();
            }
        }
```

Ein Datenpaket (8 Zeichen) wurde empfangen:

```
        private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
```

Inhalt des Empfangspuffers wird ausgelesen:

```
            string data = serialPort1.ReadExisting();
```

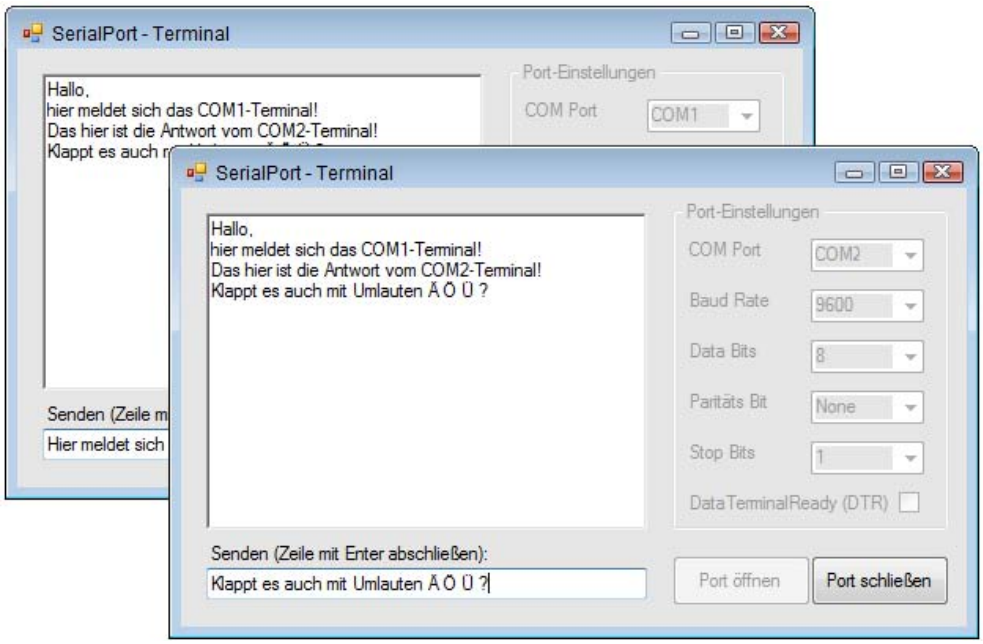
Umschalten auf den UI-Thread:

```
            richTextBox1.Invoke(new EventHandler(delegate
            {
                richTextBox1.SelectedText = string.Empty;
                richTextBox1.SelectionColor = Color.Blue;
                richTextBox1.AppendText(data);             // Anzeige
                richTextBox1.ScrollToCaret();              // Bildlauf
            }
            ));
        }
    }
}
```

Test

Verbinden Sie COM1 und COM2 Ihres Computers mittels Nullmodem-Kabel und starten Sie zwei Instanzen des Programms. Stellen Sie bei der einen Instanz den Port COM1 und bei der anderen Instanz den Port COM2 ein. Die übrigen Einstellungen sollten identisch sein (z.B. Baudrate 9600, 8 Datenbits, kein Paritätsbit, ein Stopp-Bit).

Nach Klick auf den Button "Port öffnen" können beide Terminals jetzt miteinander kommunizieren, wobei hier gesendeter Text in grüner und empfangener Text in blauer Farbe in der *RichText-Box* protokolliert werden (leider sind die farblichen Unterschiede in der Abbildung hier im Buch nicht zu erkennen):



Bemerkungen zur Verarbeitung von Messdaten

In unserem Terminalprogramm hatten wir die *ReadExisting*-Methode verwendet, um den Inhalt des Empfangspuffers komplett als Zeichenkette auszulesen. Für die Übertragung von Zahlenwerten (Messdaten etc.) scheint es aber zweckdienlicher, wenn wir von der Text- auf die Byte-Ebene hinabsteigen.

HINWEIS: In unserem [Visual C# 2012 Kochbuch] wird am Beispiel der seriellen Schnittstelle und eines Digitalvoltmeters demonstriert, wie Messdaten eingelesen und verarbeitet werden können.

28.5 Praxisbeispiele

28.5.1 Zugriff auf die Registry

Nach all dem theoretischen Geplänkel im Abschnitt 28.2 möchten Sie auch Taten sehen, wenden wir uns also einem kleinen Beispielprogramm zu.

Oberfläche

Lediglich ein Formular und drei *Button*s:

Quelltext

Einbinden eines zusätzlichen Namespace:

```
...
using Microsoft.Win32;

public partial class Form1 : Form
{
   ...
```

Das Speichern der Werte:

```
    private void Button1_Click(object sender, System.EventArgs e)
    {
       RegistryKey reg, key;
```

Öffnen des Schlüssels *Software*:

```
       reg = Registry.CurrentUser.OpenSubKey("Software", true);
```

Erzeugen des neuen Untereintrags:

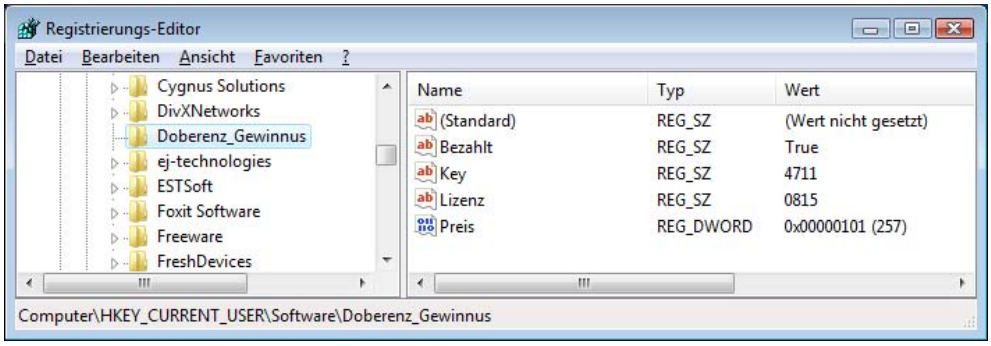
```
       key = reg.CreateSubKey("Doberenz_Gewinnus");
```

Speichern von Werten:

```
       key.SetValue("Lizenz", "0815");
       key.SetValue("Key", "4711");
       key.SetValue("Preis", 257);
```

```
        key.SetValue("Bezahlt", true);
    }
```

Das Ergebnis in der Registry:



Einlesen der Werte:

```
    private void Button2_Click(object sender, System.EventArgs e)
    {
      RegistryKey reg;
      object wert;
```

Öffnen des Baumzweigs (Sie können auch gleich mehrere Keys angeben):

```
      reg = Registry.CurrentUser.OpenSubKey("Software\\Doberenz_Gewinnus", false);
```

Auslesen zweier vorhandener Werte:

```
      wert = reg.GetValue("Bezahlt");
      MessageBox.Show(wert.ToString());
      wert = reg.GetValue("Preis");
      MessageBox.Show(wert.ToString());
```

Auslesen eines nicht vorhandenen Wertes (in diesem Fall wird der angegebene Defaultwert zurückgegeben)[1]:

```
      wert = reg.GetValue("Verkauft", false);
      MessageBox.Show(wert.ToString());
    }
```

Das Löschen eines Werts:

```
    private void Button3_Click(object sender, System.EventArgs e)
    {
      RegistryKey reg;
      reg = Registry.CurrentUser.OpenSubKey("Software\\Doberenz_Gewinnus", true);
      reg.DeleteValue("Lizenz");
      reg.Close();
```

[1] Diese Option ist beim Abfragen von Programmeinstellungen nützlich, am Anfang werden Defaultwerte eingelesen, später kann der Nutzer diese ändern.

Das Löschen eines Baumzweigs:

```
      Registry.CurrentUser.OpenSubKey("Software",
              true).DeleteSubKeyTree("Doberenz_Gewinnus");
    }
}
```

Test

Starten Sie das Programm und testen Sie zunächst die beiden ersten Funktionen. Überprüfen Sie anschließend, ob die Werte auch korrekt in die Registry eingetragen wurden.

28.5.2 Dateiverknüpfungen erzeugen

Unterstützt ein von Ihnen erstelltes Programm einen bestimmten Dateityp (.DB, .XLS etc.) ist es sinnvoll, wenn Sie dem Anwender ein entsprechendes Kontextmenü zur Verfügung stellen. Unterstützen können Sie

- *Open* (Öffnen),
- *Print* (Drucken),
- *Print To*,

... indem Sie die Registry um entsprechende Einträge erweitern. Die Variante "Print To" haben Sie sicherlich noch in keinem Kontextmenü gefunden, handelt es sich doch um die Aktion die ausgeführt wird, wenn ein Dokument mittels Drag & Drop auf ein Druckersymbol gezogen wird.

Folgende Einträge müssen Sie in der Registry vornehmen (Beispiel Metafiles):

```
01: HKEY_CLASSES_ROOT\.wmf = metafile
02: HKEY_CLASSES_ROOT\metafile = Windows Metafile
03: HKEY_CLASSES_ROOT\metafile\DefaultIcon = c:\Test\Test.exe,0
04: HKEY_CLASSES_ROOT\metafile\shell\open\command = Test.exe %1
```

```
05: HKEY_CLASSES_ROOT\metafile\shell\print\command = Test.exe /p %1
06: HKEY_CLASSES_ROOT\metafile\shell\printto\command = Test.exe /p %1
```

Zeile 1 definiert den Zusammenhang zwischen Extension und Registry-Einträgen. Zeile 2 stellt eine kurze Beschreibung des Eintrags dar. Die Angabe *DefaultIcon* ist optional, es handelt sich um den Index des Icons, das dem Dokument zugeordnet wird. Der Eintrag "...\shell\open\command" beschreibt die Aufrufkonventionen für die Anzeige des Dokuments. Analog dazu werden mit "...\print\command" bzw. "...\printto\command" die Aufrufparameter für den Druck des Dokuments festgelegt.

HINWEIS: Bei allen Einträgen sind die Werte im Feld "Default" gespeichert, es gibt keine weiteren Feldeinträge.

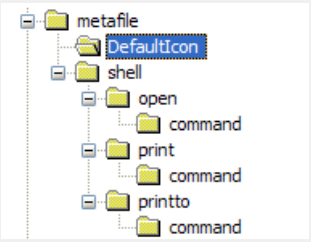
BEISPIEL 28.13: Umsetzung des obigen Beispiels als C#-Programm

C#

```
...
using Microsoft.Win32;

        private void button1_Click(object sender, EventArgs e)
        {
            RegistryKey regist;
            RegistryKey key;
            regist = Registry.ClassesRoot.OpenSubKey("", true);
            key = regist.CreateSubKey(".wmf");
            key.SetValue("", "metafile");
            key = regist.CreateSubKey("metafile");
            key.SetValue("", "Windows Metafile");
            key = regist.CreateSubKey(@"metafile\DefaultIcon");
            key.SetValue("", Application.ExecutablePath + ",0");
            key = regist.CreateSubKey(@"metafile\shell\open\command");
            key.SetValue("", Application.ExecutablePath+" %1");
            key = regist.CreateSubKey(@"metafile\shell\print\command");
            key.SetValue("", Application.ExecutablePath+" /p %1");
            key = regist.CreateSubKey(@"metafile\shell\printto\command");
            key.SetValue("", Application.ExecutablePath+" /p %1");
        }
```

Ergebnis

Das Resultat in der Registry:



Wenn Sie die Funktionen ausprobieren und die Ergebnisse mit dem Registrierungseditor kontrollieren wollen, müssen Sie die Anzeige nach jeder der oben genannten Funktionen aktualisieren (*F5).*

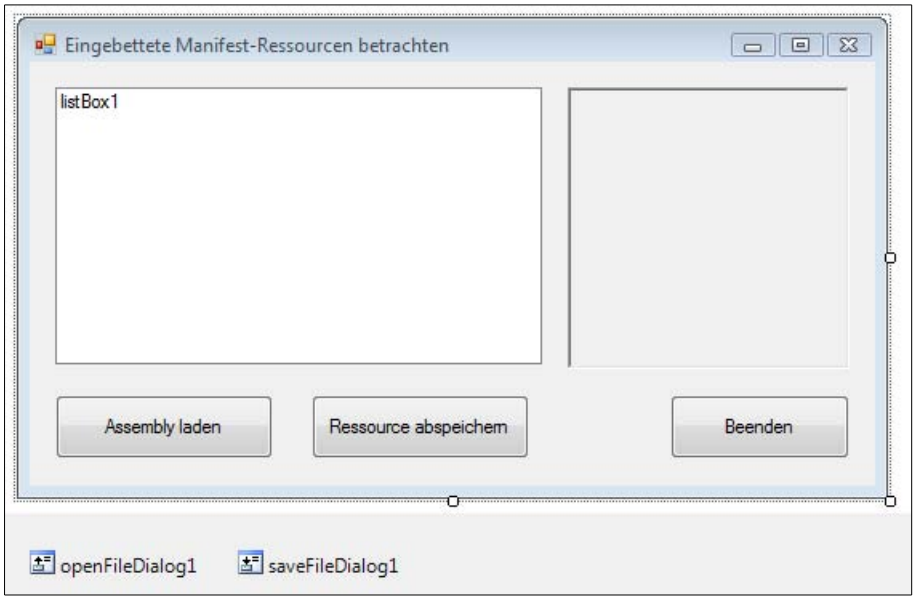
28.5.3 Betrachter für Manifestressourcen

Manifestressourcen sind beliebige Dateien, die zum Projekt hinzugefügt wurden und deren *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource* gesetzt wurde. Ein typisches Beispiel dafür sind Bilddateien.

Im vorliegenden Beispiel werden wir die *GetManifestResourceNames*- und *GetManifestResourceStream*-Methoden der *Assembly*-Klasse verwenden, um alle eingebetteten Ressourcen einer bestimmten Assembly aufzulisten und als Datei abzuspeichern. Als weiteres Feature wollen wir den Inhalt von Bild-Ressourcen anzeigen.

Oberfläche

Auf dem Startformular *Form1* finden eine *ListBox*, eine *PictureBox* und drei *Button*s ihren Platz. Außerdem brauchen wir noch je eine *OpenFileDialog*- und *SaveFileDialog*-Komponente.



Quellcode Form1

```
...
using System.Reflection;
using System.IO;
...
```

```
public class Form1 : System.Windows.Forms.Form
{
...
    private Assembly ass = null;        // globaler Zeiger auf geladene Assembly
```

Alle Ressourcen der Assembly laden, in einem Stringarray ablegen und in der *ListBox* anzeigen:

```
    private void loadResources()
    {
     string [] resNames = ass.GetManifestResourceNames();
     listBox1.Items.Clear();
     if( resNames.Length > 0 )
     {
       listBox1.BeginUpdate();
       foreach(string resName in resNames)
         listBox1.Items.Add(resName);
       listBox1.EndUpdate();
     }
     Image img = pictureBox1.Image;
     pictureBox1.Image = null;
     if( img != null )
     {
       img.Dispose();
       img = null;
     }
    }
```

Irgendeine Assembly laden:

```
        private void button1_Click(object sender, EventArgs e)
        {
            openFileDialog1.FileName = "";
            if (openFileDialog1.ShowDialog() == DialogResult.Cancel) return;
            try
            {   ass = Assembly.LoadFrom(openFileDialog1.FileName);  }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
                return;
            }
            loadResources();
        }
```

Eine Ressource in der *ListBox* auswählen, falls eine Bilddatei enthalten ist, wird deren Inhalt angezeigt:

```
        private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (listBox1.SelectedIndex < 0) return;
            Stream stream = null;
            try
```

```
            {
                stream = ass.GetManifestResourceStream((string) listBox1.SelectedItem);
                Image img = Image.FromStream(stream);
                Image oldImage = pictureBox1.Image;
                pictureBox1.Image = img;
                if (oldImage != null)      // alte Bildressource freigeben
                {
                    oldImage.Dispose();
                    oldImage = null;
                }
            }
            catch  {   }
            finally
            {  if (stream != null) stream.Close(); }
        }
```

Ressource als separate Datei abspeichern:

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (listBox1.SelectedIndex < 0)  return;
            saveFileDialog1.FileName = (string) listBox1.SelectedItem;
            if (saveFileDialog1.ShowDialog() == DialogResult.Cancel)  return;
            Stream outFile = saveFileDialog1.OpenFile();
            Stream inFile = ass.GetManifestResourceStream((string) listBox1.SelectedItem);
            long len = inFile.Length;
            if (len > int.MaxValue)
            {
                MessageBox.Show("Kann Datei in dieser Version leider nicht abspeichern!");
                outFile.Close();
                inFile.Close();
            }
            byte[] bytes = new byte[len];
            inFile.Read(bytes, 0, (int) len);
            outFile.Write(bytes, 0, (int) len);
            inFile.Close();
            outFile.Close();
        }
        ...
}
```

Test

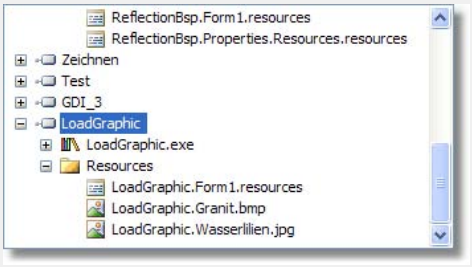
Laden Sie irgendeine Assembly (möglichst eine mit eingebetteten Grafikressourcen). Nach einem Doppelklick auf den in der *ListBox* angezeigten Namen der Ressource wird diese – sofern es sich um eine Grafikressource handelt – in der *PictureBox* angezeigt. Speichern Sie die Ressource als Datei ab.

28.5.4 Ressourcen mit Reflection auslesen

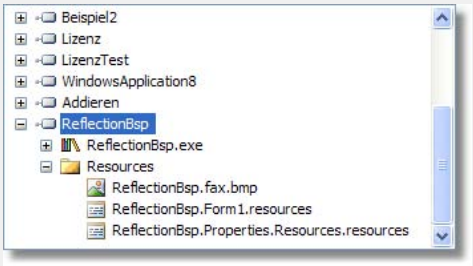
Für den Zugriff auf Ressourcen wie Grafiken, Videos, Sound etc. bietet sich die *GetManifestResourceStream*-Methode an.

Doch ach, ab Version 2.0 des Frameworks hat sich auch hier eine kleine Änderung eingeschlichen, wie eine Kontrolle mit dem .NET-Reflector zeigt:

BEISPIEL 28.14: .NET 1.x, (eingelagerte Ressourcen werden direkt unter Resources gespeichert, z.B. Wasserlilien.jpg):



BEISPIEL 28.15: Ab .NET 2.x (das Verhalten für eingelagerte Ressourcen hat sich zwar nicht geändert, Ressourcen, die über die Projekt-Eigenschaften hinzugefügt wurden, sind aber jetzt im Stream *...Properties.Resources.resources* gespeichert):



Wir müssen uns deshalb auch beim Auslesen der Ressourcen etwas mehr bemühen als bisher.

BEISPIEL 28.16: Alle Ressourcen ermitteln

C#

```
...
using System.Reflection;
using System.IO;
using System.Resources;
using System.Collections;
```

Assembly laden:

```
myass = Assembly.GetExecutingAssembly();
foreach (String s in myass.GetManifestResourceNames())
{
```

Hier bestimmen wir zunächst die einzelnen Ressource-Streams:

```
listBox1.Items.Add(s);
if (s.ToLower().EndsWith(".resources"))
```

Wenn in diesem *Stream* weitere Ressourcen enthalten sind:

```
{
  Stream stream = myass.GetManifestResourceStream(s);
  ResourceReader Reader = new ResourceReader(stream);
  IDictionaryEnumerator id = Reader.GetEnumerator();
  while (id.MoveNext())
  {
```

ID.key bezeichnet die gleiche Ressource, die Sie auch mit *Properties.Resources.xyz* auslesen können:

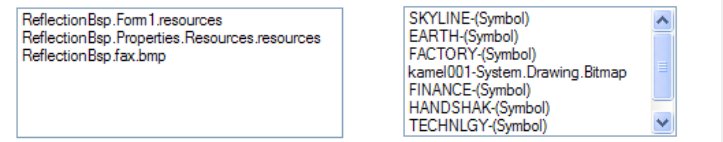
```
listBox2.Items.Add(id.Key + "-" + id.Value);
```

Über *id.Value* können wir direkt auf die einzelnen Bitmaps zugreifen:

```
if (id.Value is Bitmap)
{
    Bitmap bmp = (Bitmap) (id.Value as Bitmap).Clone();
    bmp.Save(id.Key + ".bmp");
}
}
Reader.Close();
}
}
```

Ergebnis

Die Anzeige in *listBox1* und *listBox2*:



HINWEIS: Nach dem Ausführen obigen Beispiels, werden alle Bitmaps aus der Assembly extrahiert.

BEISPIEL 28.17: Auslesen der eingebetteten Ressource *fax.bmp* und speichern in einer externen Datei bei Doppelklick auf den entsprechenden Eintrag in *listBox1*

C#
```
private void listBox1_MouseDoubleClick(object sender, MouseEventArgs e)
{
    Bitmap  bmp = new Bitmap(
    Assembly.GetExecutingAssembly().GetManifestResourceStream((String)
                      listBox1.SelectedItem));
    bmp.Save((string) listBox1.SelectedItem);
}
```

HINWEIS: Achten Sie beim Zugriff auf die Ressourcen peinlichst auf die korrekte Schreibweise (Groß-/Kleinschreibung!).

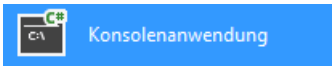
Kapitel 29

Konsolenanwendungen

Auch in Zeiten multimedialer, bunter und verspielter Windows-Oberflächen hat die gute alte Kommandozeilen- bzw. Konsolenanwendung, wie Sie diese bereits aus DOS-Urzeiten kennen, nach wie vor ihre Daseinsberechtigung. Sei es, um ein kleines Hilfsprogramm zu schreiben, oder auch nur als Beispielprogramm, bei dem es um Inhalte und nicht um optische Effekte geht (siehe Einsteigerbeispiel 1.2).

29.1 Grundaufbau / Konzepte

Eine Konsolenanwendung erstellen Sie in Visual Studio über die gleichnamige Vorlage (Menü *Datei|Neu|Projekt...*).

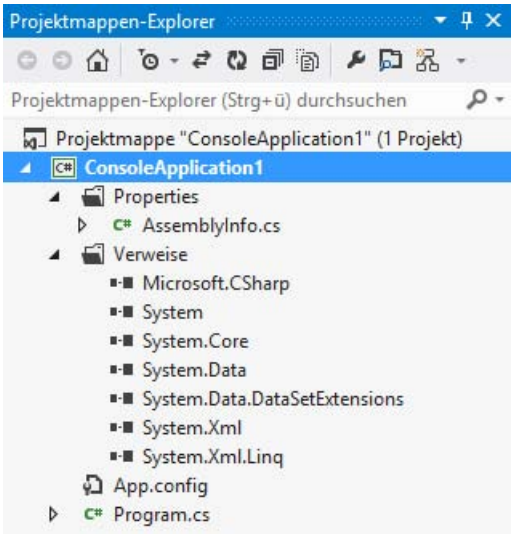


Wichtig ist in diesem Zusammenhang die Auswahl des gewünschten .NET-Zielframeworks (zur Verfügung stehen die Versionen 2, 3, 3.5, 4, 4.5), für das Sie die Konsolenanwendung entwickeln wollen.

HINWEIS: Bestimmte Anwendungs- bzw. Entwicklungsfeatures, wie z.B. das Entity-Framework, LINQ etc. sind nur verfügbar, wenn Sie eine Framework-Version wählen, die diese Features auch unterstützt. Informieren Sie sich also vorher, welche Features Sie brauchen.

Nach Abschluss des Assistenten (Sie müssen noch ein Zielverzeichnis wählen) finden Sie sich bereits im Grundgerüst Ihres zukünftigen Programms wieder. Wie schon erwähnt, steht die Oberfläche bei Konsolenanwendungen nicht im Mittelpunkt und so verwundert es auch nicht, wenn Sie es in diesem Fall mit dem C#-Quellcode von *Program.cs* zu tun bekommen (siehe folgender Abschnitt).

Ganz nebenbei werfen wir noch einen Blick auf den Projektmappen-Explorer (siehe folgende Abbildung), dieser zeigt uns noch eine weitere Datei (*AssemblyInfo.cs*), mit der wir uns ebenfalls kurz beschäftigen wollen.



Die Konsolenanwendung ist zu diesem Zeitpunkt bereits ausführbar, nach Betätigen der F5-Taste können Sie sich davon überzeugen. Allerdings werden Sie nicht mehr als ein kurzzeitig aufflackerndes Konsolen-Fenster sehen, da keinerlei Ein- und Ausgaben vorgenommen werden.

29.1.1 Unser Hauptprogramm – Program.cs

Mit der schon geöffneten Datei *Program.cs* liegt bereits das Hauptprogramm und damit der Einstiegspunkt vor Ihnen. Der C#-Code enthält neben diversen Namespace-Importen eine Klasse *Program* mit der statischen Eintrittsmethode *Main*.

BEISPIEL 29.1: Ein Hauptprogramm

C#
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        { }
    }
}
```

static void Main()

Die *Main*-Methode ist sowohl für den Start als auch für das Ende Ihres Programms verantwortlich. Aus dieser Methode heraus erstellen Sie Ihre Objekte und rufen die gewünschten Methoden auf. Ist der in der Methode definierte Code abgearbeitet, wird das Programm beendet.

Im Gegensatz zu den Windows Forms-/WPF-Anwendungen müssen wir uns im Falle einer Kommandozeilenanwendung um die "Nachrichtenschleife", d.h. in diesem Fall die dauernde Abfrage der Eingabe, selbst kümmern. Bezogen auf Beispiel 29.1 bedeutet dies, dass die Anwendung sofort nach dem Start auch schon wieder beendet ist, da weder Daten abgefragt noch ausgegeben wurden.

BEISPIEL 29.2: Ein- und Ausgabe

C#

```
...
        static void Main(string[] args)
        {
            string eingabe = "";
```

Abfragen einer Eingabe, die mit der Enter-Taste abgeschlossen wird:

```
            eingabe = Console.ReadLine();
```

Ausgabe des eingegebenen Wertes:

```
            Console.WriteLine("Sie haben '{0}' eingegeben!", eingabe);
```

Und hier warten wir auf eine beliebige Taste, sonst ist das Programm sofort beendet, ohne dass Sie die Ausgaben sehen können.

```
            Console.ReadKey();
        }
...
```

HINWEIS: Rufen Sie obiges Beispiel direkt von der Kommandozeile aus auf, können Sie auf das *ReadKey* verzichten, da das Kommandozeilenfenster ja geöffnet bleibt.

29.1.2 Rückgabe eines Fehlerstatus

Gerade Kommandozeilenprogramme werden häufig aus Skript-Dateien (.BAT, .CMD) heraus aufgerufen. In diesen Fällen steht meist das Problem, wie Programmergebnisse/-fehler sich auf den weiteren Ablauf des Skripts auswirken sollen. Auch dafür bietet die *Main*-Methode eine Lösung. Neben der bereits vorgestellten Variante ohne Rückgabewert können Sie diese Methode alternativ mit einem *int*-Rückgabewert deklarieren. Dieser kann später z.B. in einer .BAT-Datei per %ERRORLEVEL% ausgelesen und verarbeitet werden.

BEISPIEL 29.3: Rückgabe und Auswertung eines int-Wertes

C#

```
...
namespace Test
{
```

BEISPIEL 29.3: Rückgabe und Auswertung eines int-Wertes

```
    class Program
    {
        static int Main(string[] args)
        {
            return 7;
        }
    }
}
```

Die *Test.bat*:

```
rem test.bat
@echo off
test.exe
echo Rueckgabewert = %ERRORLEVEL%
```

Ergebnis

Nach dem Starten der *Test.bat* erhalten Sie folgende Ausgabe an der Kommandozeile:

HINWEIS: Über entsprechende *if*-Abfragen lässt sich so der Programmablauf des Skripts aus der Kommandozeilenanwendung heraus steuern.

29.1.3 Parameterübergabe

Kommandozeilenprogramme erhalten in vielen Fällen die nötigen Informationen als Parameter beim Programmstart. Wie Sie sicher schon vermutet haben, übernimmt der String-Array-Parameter *args* der *Main*-Methode die Aufgabe, die übergebenen Parameter abzubilden.

BEISPIEL 29.4: Auswerten der Kommandozeilenparameter

C#

```
        static void Main(string[] args)
        {
            foreach (string arg in args)
                Console.WriteLine(arg);
            Console.ReadKey();
        }
```

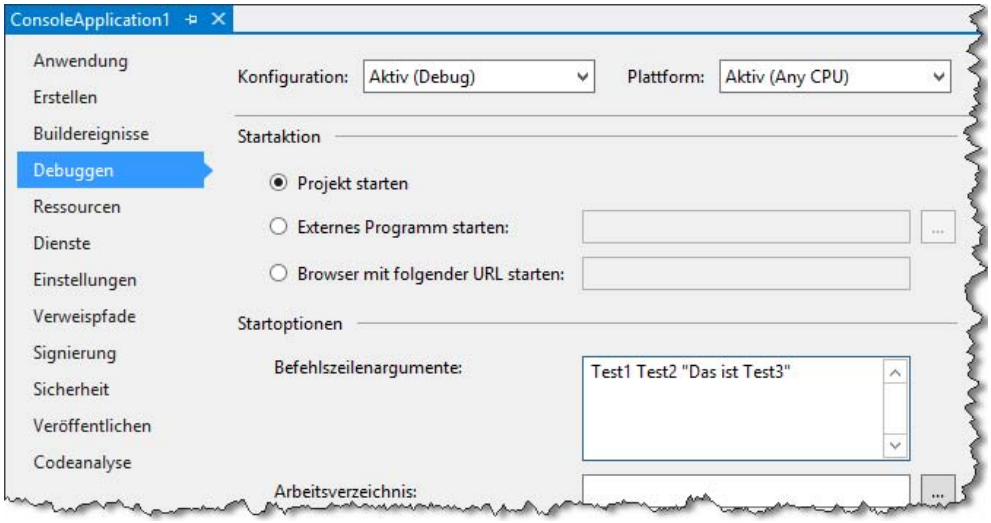
Alternativ können Sie auch die *Environment.GetCommandLineArgs*-Collection auswerten, wie es folgendes Beispiel zeigt:

BEISPIEL 29.5: Auswerten der Kommandozeilenparameter per *GetCommandLineArgs*

C#
```
static void Main(string[] args)
{
    foreach (string arg in Environment.GetCommandLineArgs())
        Console.WriteLine(arg);
    Console.ReadKey();
}
```

HINWEIS: Beachten Sie, dass diese Variante im ersten Element den kompletten Pfad der EXE zurückgibt!

Um beim Testen nicht jedes Mal zur Kommandozeile wechseln zu müssen, können Sie die Parameter auch über die Visual Studio-IDE übergeben.



Rufen Sie dazu über *Projekt|Eigenschaften|Debuggen* den obigen Dialog auf und tragen Sie die Werte in die Textbox "Befehlszeilenargumente" ein.

29.1.4 Zugriff auf die Umgebungsvariablen

Was für den Windowsprogrammierer die Registrierdatenbank ist, sind für den Entwickler von Kommandozeilentools die Umgebungsvariablen. Über diese lässt sich recht bequem der Ablauf eines kleinen Programms steuern. Abrufen können Sie die Umgebungsvariablen über die *Environment.GetEnvironmentVariables*-Methode, diese gibt ein *Dictionary* zurück, das aus einer Kombination vom *Key* und *Value* besteht.

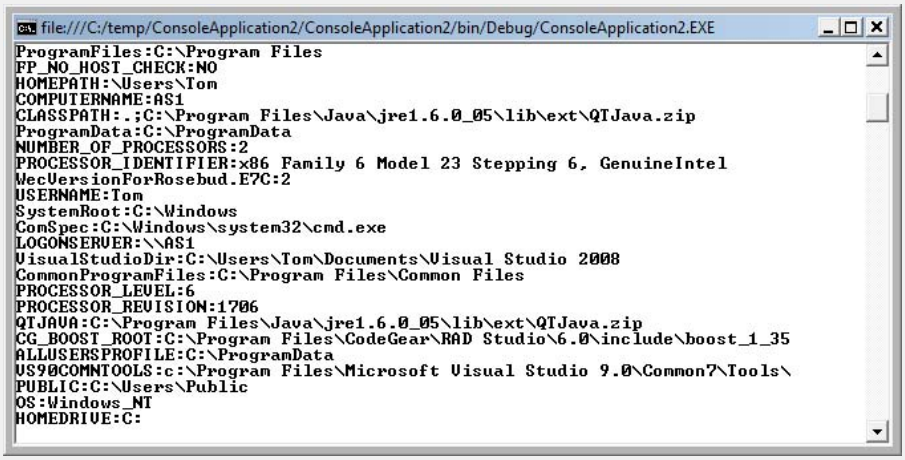
BEISPIEL 29.6: Abrufen aller Umgebungsvariablen

C#

```
static void Main(string[] args)
{
  foreach (System.Collections.DictionaryEntry de in
                  Environment.GetEnvironmentVariables())
      Console.WriteLine("{0}:{1}",de.Key, de.Value);
  Console.ReadKey();
}
```

Ergebnis

Das Ergebnis zeigt die folgende Abbildung:



Den direkten Zugriff auf eine einzelne Umgebungsvariable erhalten Sie über die Methode *GetEnvironmentVariable,* die den zugehörigen Wert zurückgibt:

SYNTAX: `public static string Environment.GetEnvironmentVariable(<Name der Variablen>)`

BEISPIEL 29.7: Abfrage der PATH-Variablen

C#

```
static void Main(string[] args)
{
    Console.WriteLine( Environment.GetEnvironmentVariable("PATH"));
    Console.ReadKey();
}
```

29.2 Die Kommandozentrale: System.Console

Haben Sie die vorhergehenden Beispiele aufmerksam studiert, wird Ihnen sicher noch die Klasse *System.Console* in Erinnerung geblieben sein. Diese Klasse enthält zahlreiche Mitglieder, die zum einen die Ein- und Ausgabe, zum anderen aber auch das optische Erscheinungsbild der Kommandozeile bestimmen. So können Sie unter anderem Größe und Position des Konsolenfensters und

dessen Farbeinstellungen verändern, Ausgaben absolut positionieren, den Kursor ein-/ausblenden, Signaltöne ausgeben usw. Die folgenden Tabellen und Beispiele sollen einen ersten Überblick bieten.

29.2.1 Eigenschaften

Wichtige Eigenschaften von *Console*:

Eigenschaft	Typ	Beschreibung
Title	*string*	Titel des Konsolenfensters
CursorVisible	*bool*	Sichtbarkeit des Kursors
CursorTop	*int*	Zeilenposition des Kursors
CursorLeft	*int*	Spaltenposition des Kursors
BackgroundColor	*ConsoleColor*	Hintergrundfarbe der Schrift
ForegroundColor	*ConsoleColor*	Vordergrundfarbe der Schrift
BufferHeight, BufferWidth	*int*	Höhe und Breite des Pufferbereichs
CapsLock	*bool*	Ist die Feststelltaste gedrückt?
Error	*TextWriter*	Standard-Fehlerausgabestream
In	*TextReader*	Standard-Eingabestream
Out	*TextWriter*	Standard-Ausgabestream
WindowHeight,WindowWidth	*int*	die Fensterabmessungen
WindowTop, WindowLeft	*int*	Position des Fensterbereichs in Bezug auf den Bildschirmpuffer

29.2.2 Methoden/Ereignisse

Wichtige Methoden von *Console*:

Methode	Beschreibung
Clear	... löscht das Konsolenfenster
SetWindowSize	... setzt Höhe und Breite des Konsolenfensters
SetCursorPosition	... positioniert den Kursor
ResetColor	... setzt Farbeinstellungen auf Standardwerte
SetBufferSize	... setzt Höhe und Breite des Bildschirmpuffers
SetWindowPosition	... setzt die Position des Konsolenfensters relativ zum Bildschirmpuffer
MoveBufferArea	... verschiebt kompletten Fensterausschnitt
ReadKey	... liest nachfolgende Tastatureingabe
Read	... liest einzelnes Zeichen aus dem Eingabestream

Methode	Beschreibung
ReadLine	... liest eine Zeile aus dem Eingabestream
Beep	... erzeugt Signalton
Write	... schreibt Zeichen ohne Zeilenvorschub
WriteLine	... schreibt eine Zeile

Last but not least, verfügt *Console* auch noch über ein Ereignis, mit dem Sie auf die Tastenkombination Strg+C reagieren können:

SYNTAX: `public static event ConsoleCancelEventHandler CancelKeyPress`

Im Weiteren wollen wir an einigen Beispielen die Verwendung der obigen Member demonstrieren.

29.2.3 Textausgaben

Wie der Tabelle unter 29.2.2 zu entnehmen ist, bieten sich für die Ausgabe von Text die Methoden *Write* und *WriteLine* an. Beide verfügen über eine beeindruckende Vielzahl von Überladungen, die es ermöglichen, fast jeden relevanten Datentyp ohne vorherige Konvertierung zu übergeben. Beide Methoden unterscheiden sich lediglich darin, dass *Write* auf das Anfügen eines Zeilenvorschubs verzichtet. Die Ausgabe selbst erfolgt an der aktuellen Position, die Sie wiederum mit *CursorLeft* und *CursorTop* beeinflussen können.

BEISPIEL 29.8: Verwendung *Write/WriteLine*

C#

```
static void Main(string[] args)
{
    Console.Write("H");
    Console.Write("a");
    Console.Write("l");
    Console.Write("l");
    Console.Write("o");
    Console.Write(" ");
    Console.WriteLine("Welt");
    Console.CursorTop = 5;
    Console.CursorLeft = 10;
    Console.Write("Absolute Positionierung");
    Console.ReadKey();
}
```

Ergebnis

HINWEIS: Über die Eigenschaft *Console.Out.NewLine* haben Sie Einfluss auf die Art des Zeilenvorschubs, so können Sie den Wert beispielsweise auf "\r\n\r\n" festlegen, um einen doppelten Zeilenvorschub zu erzielen.

29.2.4 Farbangaben

Für die optische "Verbesserung" Ihrer Anwendung bieten sich die Eigenschaften *BackGroundColor*- und *ForeGroundColor* an. Beide Farbeigenschaften werden durch Werte aus der *ConsoleColor*-Enumeration repräsentiert. Die Farbpalette erinnert an gute alte DOS-Zeiten:

Nr	Farbe	Nr	Farbe	Nr	Farbe	Nr	Farbe
0	Black	4	DarkRed	8	DarkGray	12	Red
1	DarkBlue	5	DarkMagenta	9	Blue	13	Magenta
2	DarkGreen	6	DarkYellow	10	Green	14	Yellow
3	DarkCyan	7	Gray	11	Cyan	15	White

BEISPIEL 29.9: Verwendung von Farbwerten

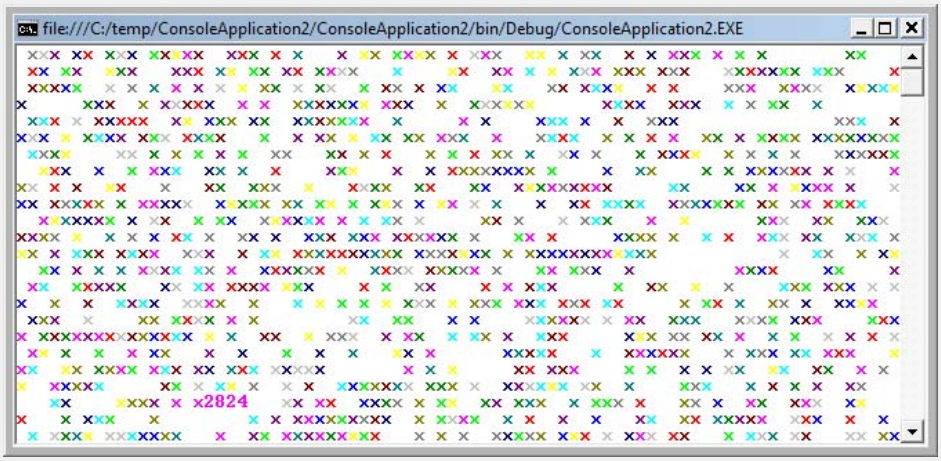
C#

Diese Konsolenanwendung verteilt zufällig verschiedenfarbige Zeichen "x" über die verfügbare Fensterfläche. Falls das gleiche Zeichen zweimal hintereinander erscheint, hält das Programm mit einem Signalton an. An der Position des letzten Zeichens wird die Gesamtanzahl von Schleifendurchläufen ausgegeben.

```
static void Main(string[] args)
{
   const int ym = 40;
   const int xm = 80;
   int xp = 0; int yp = 0;
   Console.CursorVisible = false;
   Console.SetWindowSize(xm, ym);
   System.Random rnd = new System.Random();
   int i = 0;
   do
   {
       xp = Console.CursorLeft; yp = Console.CursorTop;
       Console.CursorLeft  = rnd.Next(0, xm);
       Console.CursorTop = rnd.Next(0, ym);
       Console.ForegroundColor = (ConsoleColor) rnd.Next(1, 16);
       Console.Write("x");
       i++;
   }
   while (!((Console.CursorLeft == xp)&&(Console.CursorTop == yp)));
   Console.Beep(1000, 3000);
   Console.Write(i.ToString());
   Console.ReadKey();
}
```

BEISPIEL 29.9: Verwendung von Farbwerten

Ergebnis



29.2.5 Tastaturabfragen

Die *ReadKey*-Methode liefert ein Objekt (genauer genommen eine Struktur) vom Typ *ConsoleKeyInfo*, dem man alle Informationen über die gedrückte Taste entnehmen kann.

Eigenschaft	Beschreibung
Key	... liefert die gedrückte Taste als Mitglied der *ConsoleKey*-Enumeration
KeyChar	... liefert das Unicode-Zeichen der gedrückten Taste
Modifiers	... liefert die gleichzeitig gedrückte ALT-, STRG- oder SHIFT-Taste als Mitglied der *ConsoleModifiers*-Enumeration

BEISPIEL 29.10: Verwendung von *ReadKey*

C#

Nach jeder Tastenbetätigung wird "Hallo" ausgegeben. Nur mittels Esc-Taste kann die folgende Schleife verlassen werden.

```
static void Main(string[] args)
{
    ConsoleKeyInfo cki;
    do
    {
        Console.WriteLine("Hallo");
        cki = Console.ReadKey(true);
    }
    while (cki.Key != ConsoleKey.Escape);
}
```

In der *ConsoleKey*-Enumeration ist quasi jede nur denkbare Taste enthalten (siehe Online-Dokumentation), lediglich für Shift, Alt oder Strg gibt es eine Sonderbehandlung durch die *ConsoleModifiers*-Enumeration.

Modifiers-Eigenschaft und ConsoleModifiers-Enumeration

Das *ConsoleKeyInfo*-Objekt kann mittels seiner *Modifiers*-Eigenschaft und deren bitweiser Verknüpfung mit einem Mitglied der *ConsoleModifiers*-Enumeration auch feststellen, ob zusammen mit einer anderen Taste die Shift-, Alt- oder Strg-Taste betätigt wurde, sodass quasi beliebige Tastaturabfragen möglich sind.

BEISPIEL 29.11: Es werden die gedrückte Tastenkombination und der zugehörige Unicode ausgegeben.

C#

```
ConsoleKeyInfo cki = Console.ReadKey(true);
StringBuilder sb = new StringBuilder();
sb.Append("Sie haben gedrückt die Taste(n): ");
if (cki.Modifiers != 0)
{
     if ((cki.Modifiers & ConsoleModifiers.Shift) != 0)
        sb.Append("SHIFT+");
     if ((cki.Modifiers & ConsoleModifiers.Alt) != 0)
        sb.Append("ALT+");
     if ((cki.Modifiers & ConsoleModifiers.Control) != 0)
        sb.Append("CTL+");
}
sb.Append(cki.Key.ToString() + "  (Zeichen: " + cki.KeyChar + ")");
Console.WriteLine(sb.ToString());
```

Ergebnis

Als Ergebnis erscheint auf der Konsole z.B.:

```
Sie haben gedrückt die Taste(n): SHIFT+ALT+H   (Zeichen: H)
```

HINWEIS: Wenn Sie das Beenden durch Strg+C vermeiden möchten, müssen Sie die folgende Anweisung einfügen: *Console.TreatControlCAsInput = true;*

29.2.6 Arbeiten mit Streamdaten

In alten DOS-Zeiten waren Konstrukte wie dieses häufig anzutreffen:

```
dir *.* | programmxyz.exe > ausgabe.txt
```

Sinn und Zweck eines solchen Konstrukts ist das Umleiten der Ausgabe von "dir" mittels Pipe (das |-Zeichen) an die Eingabe der Anwendung "programmxyz.exe". Die Ausgaben von "programmxyz.exe" werden wiederum in eine neue Datei "ausgabe.txt" umgeleitet (per >-Zeichen).

Möchten Sie mit C# eine derartige Anwendung erstellen, die Eingaben wie oben gezeigt entgegennimmt, verarbeitet und weitergeben kann, vergessen Sie ganz schnell die meisten der bisher

beschriebenen Features. Dreh- und Angelpunkt sind jetzt die beiden Eigenschaften *Console.In* und *Console.Out*.

HINWEIS: Ein absolutes Positionieren des Kursors oder andere derartige Anweisungen sind in diesem Zusammenhang nicht angebracht und führen zwangsläufig zu einem Laufzeitfehler.

Bei *Console.In* bzw. *Console.Out* handelt es sich um einen *TextReader* bzw. einen *TextWriter*, was die Verwendung wesentlich vereinfacht.

BEISPIEL 29.12: Verwendung von *Console.In* und *Console.Out*

C#

Einlesen des Eingabestreams und Ausgabe mit laufender Zeilennummer.

```
static int Main(string[] args)
{
    string zeile = "";
    int nr = 0;
```

Zeilenweises Verarbeiten des Eingabestreams solange Daten vorliegen:

```
    while ((zeile = Console.In.ReadLine()) != null)
    {
```

Zeilenweise Ausgabe der gelesenen Daten:

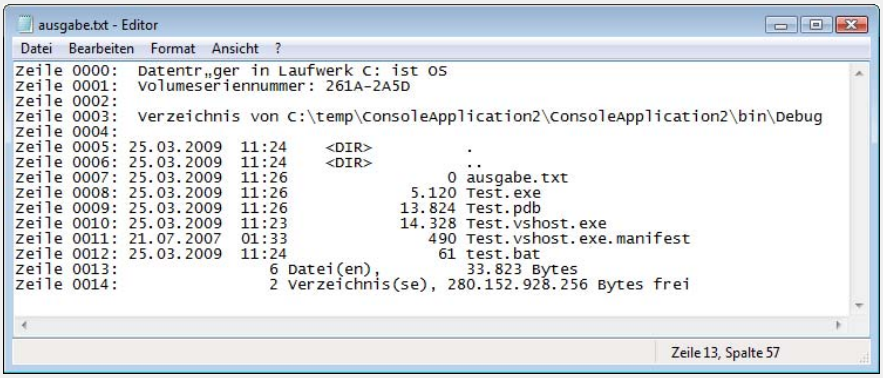
```
        Console.Out.WriteLine("Zeile {0}: {1}", nr.ToString("D4"), zeile);
        nr += 1;
    }
    return 0;
}
```

Ergebnis

Rufen Sie die erstellte Anwendung *Test.exe* an der Kommandozeile wie folgt auf:

```
dir *.* | Test.exe > ausgabe.txt
```

Jetzt können Sie das Ausgabeergebnis in der Datei *ausgabe.txt* betrachten:



29.3 Praxisbeispiel

29.3.1 Farbige Konsolenanwendung

Das folgende, etwas längere, Beispiel soll Ihnen noch einmal die diversen Features der *Console*-Klasse demonstrieren.

```
class Program
{
```

Die folgende Methode zeichnet einen "Truck" in der Farbe *cc* am linken Rand der Zeile *y*:

```
        static void drawTruck(ConsoleColor cc, int y)
        {
```

Karosserie zeichnen:

```
            Console.BackgroundColor = cc;
            Console.CursorTop = y;
            Console.CursorLeft = 5;
            Console.Write(" ");
            Console.CursorLeft = 6;
            Console.Write(" ");
            Console.CursorTop = y+1;
            for (int i = 1; i < 9; i++)
            {
                Console.CursorLeft = i;
                Console.Write(" ");
            }
```

Räder zeichnen:

```
            Console.BackgroundColor = ConsoleColor.Black;
            Console.ForegroundColor = cc;
            Console.CursorTop = y+2;
            Console.CursorLeft = 2;
            Console.Write("O");
            Console.CursorLeft = 7;
            Console.Write("O");
        }
```

Der Einsprung in das Programm:

```
        static void Main(string[] args)
        {
            const int xm = 80;       // Fensterbreite
            const int ym = 40;       //    "   höhe
            const int ypos = 5;      // Spur, in welcher Truck fährt
            string msg = "Drücken Sie eine beliebige Taste, um {0}";   // Formatzeichenkette
```

Kopfzeile festlegen:

```
            Console.Title = "Konsolen-Truck";
```

Fenstergröße bestimmen:

```
Console.SetWindowSize(xm, ym);
```

Ausgaben realisieren:

```
Console.WriteLine("Willkommen zur Truck-Demo!\n");
Console.Write("Welche Farbe soll der Truck haben? (rot=1, blau=2, grün=3) ");
```

Auf Eingabe warten:

```
string farbe = Console.ReadLine();
```

Kursor ausblenden:

```
Console.CursorVisible = false;
switch (farbe)
{
    case "1": drawTruck(ConsoleColor.Red, ypos); break;
    case "2": drawTruck(ConsoleColor.Blue, ypos); break;
    case "3": drawTruck(ConsoleColor.Green, ypos); break;
    default: drawTruck(ConsoleColor.White, ypos); break;
}
Console.SetCursorPosition(0, ypos + 4);
Console.ResetColor();
Console.WriteLine(msg, "den Truck zu starten!");
Console.ReadKey();
Console.Beep();
```

Unser Truck "fährt" jetzt von links nach rechts:

```
for (int i = 0; i < (xm - 9); i++)
{
        Console.MoveBufferArea(i, ypos, 9, 3, i + 1, ypos);
        System.Threading.Thread.Sleep(50);
}
```

Alles löschen:

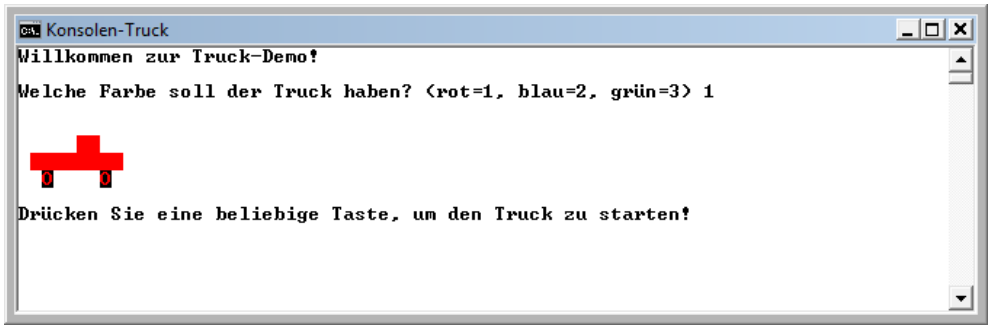
```
Console.Clear();
```

Warnton ausgeben:

```
Console.Beep();
Console.SetCursorPosition(0, 0);
Console.ResetColor();
Console.WriteLine(msg, "die Truck-Demo zu beenden!");
```

Diese Anweisung brauchen wir, sonst wird das Fenster schon hier geschlossen (Ende des Hauptprogramms:

```
            Console.ReadKey();
        }
}
```



29.3.2 Weitere Hinweise und Beispiele

Wer sich intensiver mit der Entwicklung von Konsolenanwendungen beschäftigen will, findet im Internet weitere Informationen:

Die Hilfe für die *Console*-Klasse:

LINK: `http://msdn.microsoft.com/de-de/library/system.console.aspx`

Ein Parser für Kommandozeilenparameter:

LINK: `http://www.codeproject.com/KB/recipes/command_line.aspx`

Damit wollen wir das Thema "Konsolenanwendungen" abschließen, denn das Haupteinsatzgebiet für Ihre C#-Anwendungen dürfte die Windows-Oberfläche sein.

Teil VI

Teil VI: Windows Forms

- **Windows Forms-Anwendungen**
- **Windows-Formulare verwenden**
- **Komponenten-Übersicht**
- **Einführung Grafikausgabe**
- **Druckausgabe**
- **Windows Forms-Datenbindung**
- **Erweiterte Grafikausgabe**
- **Ressourcen/Lokalisierung**
- **Komponentenentwicklung**

Kapitel 30

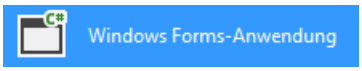
Windows Forms-Anwendungen

Nicht mehr ganz frisch, aber dennoch aktuell: die Windows Forms-Anwendungen. Diese entsprechen, im Gegensatz zu den WPF-Anwendungen, in Ihrer optischen Aufmachung weitgehend ihren Win32-Pendants. Der große Vorteil dieses Anwendungstyps ist die fast schon unüberschaubare Vielfalt an Komponenten, die es für fast jede Aufgabe gibt.

Ausgerüstet mit grundlegenden C#- und OOP-Kenntnissen werden Sie die folgenden Kapitel in die Lage versetzen, umfangreichere Benutzerschnittstellen eigenständig zu programmieren. Der Schwerpunkt des aktuellen Kapitels liegt zunächst auf dem Anwendungstyp selbst, nachfolgend werden wir uns mit dem Formular als der "Mutter aller Windows Forms-Komponenten" und seiner grundlegenden Interaktion mit den Steuerelementen beschäftigen. Abschließend werfen wir einen Blick auf die einzelnen visuellen Komponenten bzw. Steuerelemente.

30.1 Grundaufbau/Konzepte

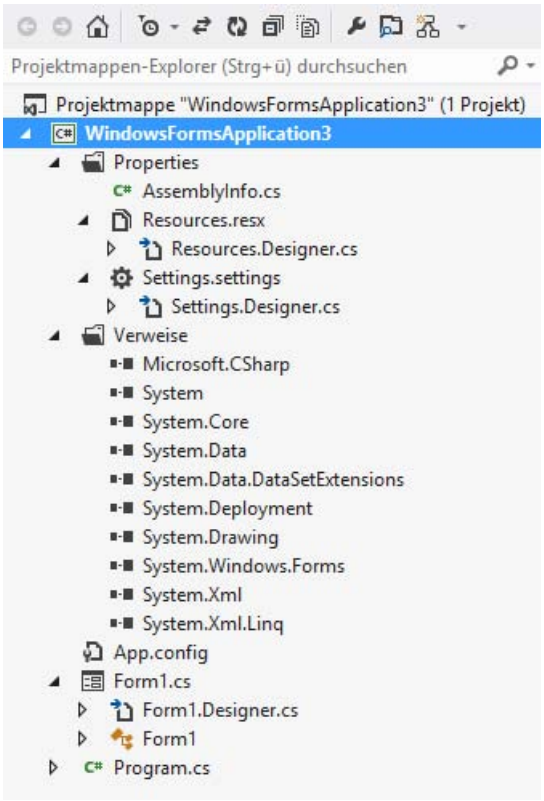
Die "gewöhnliche" Windows Forms-Anwendung erstellen Sie in Visual Studio über die gleichnamige Vorlage:



Beachten Sie in diesem Zusammenhang auch die unscheinbare kleine Combox am oberen Dialogfensterrand, über die Sie vorgeben, für welches .NET-Zielframework Sie die Anwendung entwickeln wollen.

Bestimmte Anwendungs- bzw. Entwicklungsfeatures, wie z.B. das Entity-Framework, LINQ etc., sind nur verfügbar, wenn Sie eine Framework-Version wählen, die diese Features auch unterstützt. Informieren Sie sich also vorher, welche Features Sie unterstützen müssen.

Nachdem Sie diesen Schritt erfolgreich hinter sich gebracht haben, finden Sie im Designer bereits ein einfaches Windows Form vor. Ein Blick in den Projektmappen-Explorer (siehe folgende Abbildung) zeigt jedoch noch eine Reihe weiterer Dateien, mit denen wir uns im Folgenden intensiver beschäftigen wollen.



Die Anwendung selbst ist zu diesem Zeitpunkt bereits ausführbar, ein Klick auf F5 und Sie können sich davon überzeugen. Doch nach diesem Blick auf die Bühne wollen wir nun zunächst in den "Technik-Keller" hinabsteigen, bevor wir uns in den weiteren Kapiteln mit der "Requisite", d.h. der optischen Gestaltung der Anwendung, abmühen wollen.

30.1.1 Das Hauptprogramm – Program.cs

Der Einstiegspunkt für unsere Anwendung findet sich, wie Sie es sicher schon vermutet haben, in der Datei *Program.cs*. Der bereits vorliegende Code (Beispiel 30.1) enthält neben diversen Namespace-Importen eine Klasse *Program* mit der statischen Eintrittsmethode *Main*.

BEISPIEL 30.1: *Program.cs*

C#
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
```

BEISPIEL 30.1: *Program.cs*

```
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

static void Main()

Im einfachsten Fall enthält die *Main*-Methode lediglich drei Methodenaufrufe (auf die Details kommen wir in den folgenden Abschnitten zu sprechen). Alternativ können Sie jedoch auch eine *Main*-Methode mit der Rückgabe eine *int*-Wertes und/oder der Übergabe eines *string*-Arrays realisieren (siehe dazu auch Kapitel 29, Konsolenanwendungen). Über dieses String-Array können Sie die Kommandozeilenparameter des Programms auswerten.

BEISPIEL 30.2: Auswerten von Kommandozeilenparametern

C#
```
...
        static void Main(string[] args)
        {
            foreach (string arg in args)
                Console.WriteLine(arg);
...
```

HINWEIS: Günstiger ist die Auswertung der Kommandozeilenparameter jedoch über die *GetCommandLineArgs*-Collection, diese steht jederzeit zur Verfügung und erfordert nicht schon beim Programmstart eine entsprechende Auswertung.

BEISPIEL 30.3: Verwendung von *GetCommandLineArgs*

C#
```
...
      foreach (string arg in Environment.GetCommandLineArgs())
      {
          Console.WriteLine(arg);
      }
...
```

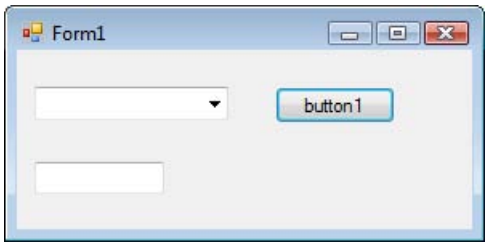
Kommen wir jetzt zum Inhalt der *Main*-Methode.

Application.EnableVisualStyles

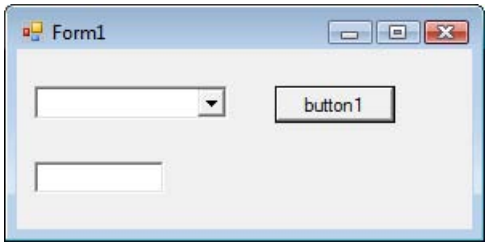
Sicher ist auch Ihnen nicht verborgen geblieben, dass sich seit Windows XP ein neues Layout für die einzelnen Windows Forms-Steuerelemente eingebürgert hat. Voraussetzung für deren Nutzung ist der entsprechende Aufruf der *Application.EnableVisualStyles*-Methode.

Wie sich der Aufruf auf ein fiktives Formular auswirkt, zeigen die beiden folgenden Abbildungen.

Mit *EnableVisualStyles*-Aufruf:



Ohne *EnableVisualStyles*-Aufruf:



Application.SetCompatibleTextRenderingDefault(false)

Mit Einführung von Windows Forms 2.0 wurde die Ausgabe von Text teilweise umgestellt. Da diese Variante nicht 100% kompatibel mit den bisherigen Ausgabeergebnissen ist, wurde für einzelne Controls eine Eigenschaft *UseCompatibleTextRendering* definiert, mit der bestimmt wird, ob die alte Variante (1.0 bzw. 1.1) oder die neue Variante (ab 2.0) verwendet werden soll. Über die Methode *Application.SetCompatibleTextRenderingDefault* wird dieser Wert für alle Controls der Anwendung voreingestellt.

Lange Rede kurzer Sinn: Ist der übergebene Wert *true,* verwenden die Steuerelemente GDI für die Textausgabe, andernfalls GDI+.

Application.Run(new Form1())

Und damit sind wir auch schon beim eigentlichen Mittelpunkt eines jeden Windows Forms-Programms angekommen. *Application.Run* erstellt die für jede Windows-Anwendung lebensnotwendige Nachrichtenschleife, die beispielsweise Maus- und Tastaturereignisse, aber auch Botschaften des Systems oder anderer Anwendungen, entgegennimmt.

Die zweite Aufgabe dieser statischen Methode ist das Instanziieren des Hauptformulars und dessen Anzeige. Wird dieses Fenster geschlossen, ist automatisch auch die Nachrichtenschleife der Anwendung beendet, das Programm terminiert. Dies gilt auch, wenn Sie aus dem Hauptformular heraus (dies ist zunächst *Form1*) weitere Formulare öffnen.

Besteht Ihre Anwendung aus mehr als nur einem Formular, stehen Sie sicher auch vor der Frage, welches Formular denn das Hauptformular bzw. das Startobjekt sein soll. Standardmäßig ist nach dem Erstellen eines Windows Forms-Projekts *Form1* als Startobjekt definiert. Nachträglich können Sie jederzeit auch ein anderes Fenster oder auch eine spezielle Prozedur als Startobjekt festlegen. Durch Editieren des von Visual Studio automatisch generierten Konstruktoraufrufs zum Erzeugen des Startformulars *Form1* kann ein beliebiges anderes Formular der Anwendung zum Startformular gekürt werden.

Nicht in jedem Fall ist es jedoch erwünscht, dass gleich ein Formular beim Start der Anwendung angezeigt wird. Sollen beispielsweise Übergabeparameter ausgewertet werden, ist es häufig günstiger, zunächst diese zu bearbeiten und dann erst das geeignete Formular anzuzeigen. In diesem Fall ändern Sie einfach obigen Abschnitt entsprechend Ihren Wünschen.

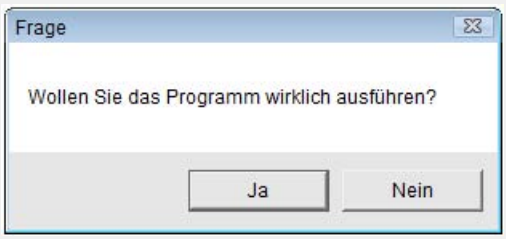
BEISPIEL 30.4: Eine *Main*-Methode, die zunächst eine *MessageBox*-Abfrage startet

C#

```
...
[STAThread]
static void Main()
{
    if (MessageBox.Show("Wollen Sie das Programm wirklich ausführen?", "Frage",
                                        MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form2());
    }
}
```

Ergebnis

Ein Start der Anwendung hat jetzt zur Folge, dass zunächst die *MessageBox*-Abfrage aufgerufen wird. Mit dem Druck auf den *Ja*-Button wird der Anwendung *Form2* als Hauptformular zugewiesen und ausgeführt. Andernfalls endet die Programmausführung an dieser Stelle.



30.1.2 Die Oberflächendefinition – Form1.Designer.cs

Eigentlich nicht für Ihre Blicke bestimmt, fristet die Datei *Form1.Designer.cs* ein relativ unbeachtetes Dasein. Doch dieser Eindruck täuscht, enthält doch diese Datei die eigentliche Klassendefinition von *Form1* inklusive Initialisierung der Oberfläche. Gerade der letzte Punkt ist besonders wichtig: Jede Komponente, die Sie im Formulardesigner in das Formular einfügen und konfigurieren, wird in dieser Klassendefinition (Methode *InitializeComponent*) instanziiert und konfiguriert.

BEISPIEL 30.5: *Form1.Designer.cs*

C#

```
namespace WindowsFormsApplication1
{
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;

        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
                components.Dispose();
            base.Dispose(disposing);
        }

        #region Vom Windows Form-Designer generierter Code
```

Die folgende Methode sollten Sie keinesfalls selbst editieren[1]:

```
        private void InitializeComponent()
        {
            this.SuspendLayout();
            //
            // Form1
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(284, 112);
            this.Name = "Form1";
            this.Text = "Form1";
            this.ResumeLayout(false);
```

Ab hier folgen später die Initialisierungen der einzelnen Formular-Komponenten:

```
            ...
        }
        #endregion
    }
}
```

[1] Mit einer Ausnahme: Haben Sie später einmal einen oder mehrere Eventhandler in der Datei *Form1.cs* gelöscht, können Sie deren Zuweisung zu den einzelnen Ereignissen am bequemsten in dieser Datei aufheben. Die betreffenden Zeilen sind durch die "beliebte" rote Wellenlinie des Editors gekennzeichnet, es genügt, wenn Sie diese Zeilen einfach löschen.

Haben Sie sich obiges Listing genauer angesehen, wird Ihnen aufgefallen sein, dass die Klasse *Form1* als *partial* deklariert ist. Dies ermöglicht es, den Code der Klassendefinition auf mehrere Dateien aufzuteilen, was in unserem Fall bedeutet, dass wir in *Form1.cs* den Rest der Klasse *Form1* vorfinden.

30.1.3 Die Spielwiese des Programmierers – Form1.cs

Jetzt kommen wir zum eigentlichen Tummelplatz für Ihre Aktivitäten als Programmierer. Visual Studio hat bereits ein kleines Codegerüst vorbereitet, das aus dem Import diverser Namespaces sowie der restlichen Klassendefinition von *Form1* inklusive Konstruktor besteht:

BEISPIEL 30.6: *Form1.cs*

C#
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
```

Der Konstruktor ist dafür verantwortlich, die Methode *InitializeComponent* aufzurufen, diese initialisiert und parametriert die einzelnen Windows Forms-Komponenten:

```
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Fügen Sie später in *Form1* eine Schaltfläche (*Button*) ein, wird die Oberflächendefinition in der Datei *Form1.Designer.cs* abgelegt, den Code für die Ereignisbehandlung definieren Sie in *Form1.cs*.

BEISPIEL 30.7: Code nach Einfügen eines Buttons

C#
In *Form1.Designer.cs*:

```
        private void InitializeComponent()
        {
            this.button1 = new System.Windows.Forms.Button();
            this.SuspendLayout();
```

BEISPIEL 30.7: Code nach Einfügen eines Buttons

```
            //
            // button1
            //
            this.button1.Location = new System.Drawing.Point(55, 35);
            this.button1.Name = "button1";
            this.button1.Size = new System.Drawing.Size(75, 23);
            this.button1.TabIndex = 0;
            this.button1.Text = "button1";
            this.button1.UseVisualStyleBackColor = true;
            this.button1.Click += new System.EventHandler(this.button1_Click);
...
```

In *Form1.cs*:

```
        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }
```

30.1.4 Die Datei AssemblyInfo.cs

Verschiedene allgemeine Informationen, wie z.B. der Titel der Anwendung oder das Copyright, können als Attribute in die Datei *AssemblyInfo.cs* eingetragen werden. Zugriff auf diese Datei erhalten Sie über den Projektmappen-Explorer (unter *Properties*).

BEISPIEL 30.8: Auszug aus *AssemblyInfo.cs*

C#

```
...
[assembly: AssemblyTitle("WindowsFormsApplication1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("WindowsFormsApplication1")]
[assembly: AssemblyCopyright("Copyright ©  2009")]
[assembly: AssemblyCulture("")]
...
```

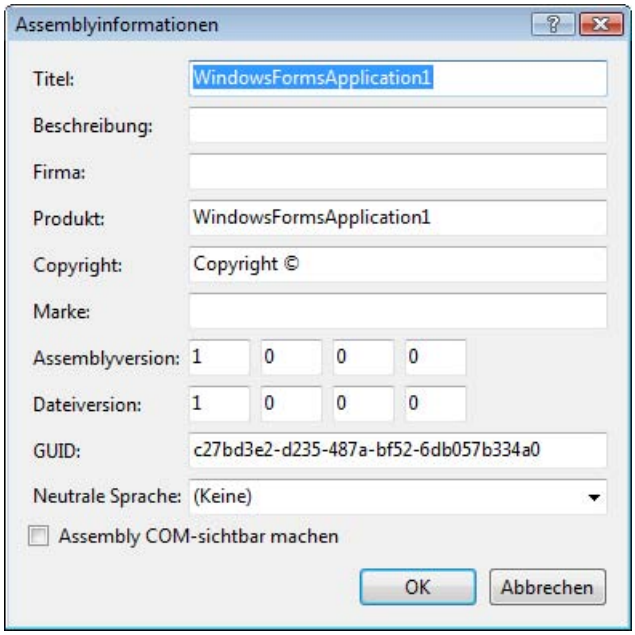
Zur Laufzeit können Sie auf die Einträge über bestimmte Eigenschaften des *Application*-Objekts zugreifen.

BEISPIEL 30.9: Anzeige des Eintrags unter *AssemblyProduct* in einem Meldungsfenster

C#

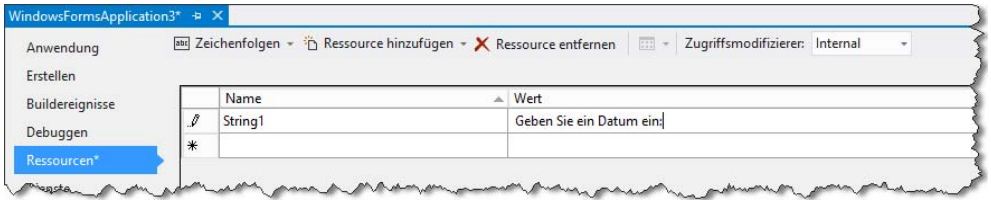
```
MessageBox.Show(Application.ProductName);
```

Editieren lassen sich die Einträge am einfachsten über den folgenden Dialog, den Sie über *Projekt*|*Eigenschaften*|*Anwendung*|*Assemblyinformationen* erreichen:



30.1.5 Resources.resx/Resources.Designer.cs

Beide Dateien stellen das Abbild der unter *Projekt|Eigenschaften|Ressourcen* definierten Ressourcen dar. Die XML-Datei *Resources.resx* enthält die eigentlichen Definitionen, die Sie mit dem in der folgenden Abbildung gezeigten Editor erstellen können.



Hier können Sie sehr komfortabel nahezu beliebige Ressourcen (Bilder, Zeichenketten, Audiodateien, ...) zu Ihrem Projekt hinzufügen.

Im Gegensatz dazu stellt die Datei *Resources.Designer.cs* für die Ressourcen entsprechende Mapperklassen bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfachen und typisieren. Verantwortlich dafür ist die *Properties.Resources*-Klasse:

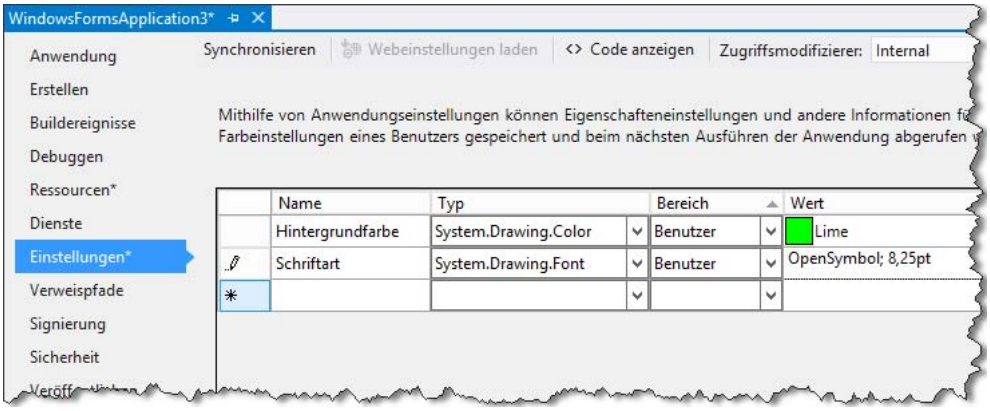
SYNTAX: `<Projekt Namespace>.Properties.Resources.<ResourceName>`

BEISPIEL 30.10: Die Bildressource *BeimChef.bmp* wird in einer *PictureBox* angezeigt

C#
```
pictureBox1.Image = Properties.Resources.BeimChef;
```

30.1.6 Settings.settings/Settings.Designer.cs

Diese beiden Dateien sind zur Entwurfszeit für die Verwaltung der Programmeinstellungen verantwortlich. Die XML-Datei *Settings.Settings* enthält die eigentlichen Werte, die Sie jedoch nicht direkt zu bearbeiten brauchen, sondern unter Verwendung eines komfortablen Editors:



Für jede Einstellung können Sie *Name*, *Typ*, *Bereich* und *Wert* festlegen. Wenn Sie den Bereich als *Anwendung* spezifizieren, wird diese Einstellung später (Laufzeit) in der Konfigurationsdatei *<Anwendungsname.exe>.config* unter dem *<applicationSettings>*-Knoten gespeichert. Falls Sie *Benutzer* wählen, erfolgt die Ablage unterhalb des *<userSettings>*-Knotens.

Die Datei *Settings.Designer.cs* stellt für die oben definierten Einträge eine Mapperklasse bereit, die den späteren Zugriff auf die einzelnen Einträge vereinfacht und vor allem typisiert.

BEISPIEL 30.11: *Settings.Designer.cs*

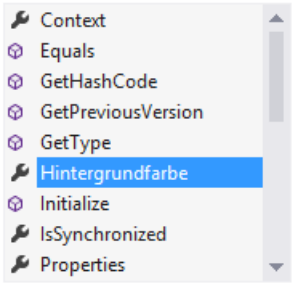
C#
```
namespace WindowsFormsApplication1.Properties {
...
internal sealed partial class Settings : global::System.Configuration.ApplicationSettingsBase
{
...
        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("Lime")]
        public global::System.Drawing.Color Hintergrundfarbe {
            get {
                return ((global::System.Drawing.Color)(this["Hintergrundfarbe"]));
            }
            set {
                this["Hintergrundfarbe"] = value;
            }
        }
...
```

Der Zugriff auf die eingetragenen Einstellungen ist – ähnlich dem Zugriff auf Ressourcen – einfach über die *Properties.Settings*-Klasse möglich.

SYNTAX: `<Projekt Namespace>.Properties.Settings.Default.<SettingsName>`

BEISPIEL 30.12: Verwendung der Einstellungen

C#
```
private void Form1_Load(object sender, EventArgs e)
{
    this.BackColor = Properties.Settings.Default.
}
```



Doch auch das Zurückschreiben von Nutzer-Einstellungen in die Datei ist über obige Klasse problemlos möglich.

BEISPIEL 30.13: Einstellungsänderungen in die Config-Datei zurückschreiben

C#
```
Properties.Settings.Default.myColor = Color.AliceBlue;
Properties.Settings.Default.Save();
```

HINWEIS: Da auch der spätere Programmbenutzer die zur Assembly mitgegebene XML-Konfigurationsdatei (z.B. *<Anwendungsname>.exe.config*) editieren kann, ergeben sich einfache Möglichkeiten für nachträgliche benutzerspezifische Anpassungen, ohne dazu das Programm erneut kompilieren zu müssen.

30.1.7 Settings.cs

Nanu, hatten wir diese Datei nicht gerade? Nein, hierbei handelt es sich quasi um den zweiten Teil der Klassendefinition von *Settings* aus *Settings.Designer.cs,* was dank partieller Klassendefinition problemlos möglich ist. Im Normalfall dürften Sie diese Datei nicht zu sehen bekommen, aber wenn Sie sich die Mühe machen, über den Designer (siehe Seite 1498) den Button *Code anzeigen* anzuklicken, findet sich kurz darauf auch diese Datei in Ihrem Projekt.

BEISPIEL 30.14: *Setting.cs*

C#
```
namespace WindowsFormsApplication1.Properties {
    internal sealed partial class Settings {
        public Settings() {
```

BEISPIEL 30.14: ***Setting.cs***

```
            // Heben Sie die  Auskommentierung der unten angezeigten Zeilen auf,
            // um Ereignishandler zum Speichern und Ändern von Einstellungen hinzuzufügen:
            // this.SettingChanging += this.SettingChangingEventHandler;
            //
            // this.SettingsSaving += this.SettingsSavingEventHandler;
            //
        }

        private void SettingChangingEventHandler(object sender,
                    System.Configuration.SettingChangingEventArgs e) {
          // Fügen Sie hier Code zum Behandeln des SettingChangingEvent-Ereignisses hinzu.
        }

        private void SettingsSavingEventHandler(object sender,
                    System.ComponentModel.CancelEventArgs e) {
          // Fügen Sie hier Code zum Behandeln des SettingsSaving-Ereignisses hinzu.
        }
    }
}
```

Mit den in dieser Datei bereitgestellten Eventhandler-Rümpfen bietet sich die Möglichkeit, auf das Laden, Ändern und Sichern von Einstellungen per Ereignis zu reagieren. Es genügt, wenn Sie die entsprechenden Kommentare im Konstruktor entfernen und den gewünschten Code in die Eventhandler eintragen.

30.2 Ein Blick auf die Application-Klasse

Die *Application*-Klasse aus dem Namespace *System.Windows.Forms* ist für den Programmierer von zentraler Bedeutung und unser Kapitel über die Grundlagen von Windows Forms-Anwendungen wäre höchst unvollständig, würden wir auf diese wichtige Klasse nur ganz am Rand eingehen. Den ersten Kontakt mit ihr hatten Sie bereits in der *Main*-Methode (siehe dazu auch Seite 1490).

Die *Application*-Klasse bietet einige interessante statische Eigenschaften und Methoden, von denen wir hier nur die wichtigsten kurz vorstellen wollen:

30.2.1 Eigenschaften

Einige wichtige Eigenschaften von *Application:*

Eigenschaft	Beschreibung
AllowQuit	... darf die Anwendung beendet werden (readonly)?
CommonAppDataPath	... ein gemeinsamer Anwendungspfad.
CommonAppDataRegistry	... ein gemeinsamer Registry-Eintrag.

Eigenschaft	Beschreibung
CompanyName	... der in den Anwendungsressourcen angegebene Firmenname.
CurrentCulture	... ein Objekt mit den aktuellen Kulturinformation (Kalender, Währung).
ExecutablePath	... der Name der Anwendung inklusive Pfadangabe.
LocalUserAppDataPath	... Anwendungspfad für die Daten des lokalen Benutzers.
OpenForms	... Collection der geöffneten Formulare.
ProductName	... der in den Ressourcen angegebene Produktname.
ProductVersion	... die in den Ressourcen angegebene Produktversion.
StartupPath	... der Anwendungspfad.
UserAppDataPath	... Pfad für die Anwendungsdaten.
UserAppDataRegistry	... Registry-Schlüssel für die Anwendungsdaten.

HINWEIS: Einige der obigen Eigenschaften lassen sich den Einträgen der Datei *AssemblyInfo.cs* zuordnen (siehe dazu Seite 1496).

BEISPIEL 30.15: Abrufen aller geöffneten Formulare

C#
```
foreach (Form f in Application.OpenForms)
{
   listBox1.Items.Add(f.Name);
}
```

BEISPIEL 30.16: Abfrage von *UserAppDataPath* und *UserAppDataRegistry*

C#
```
textBox1.Text = Application.UserAppDataRegistry.ToString();
textBox2.Text = Application.UserAppDataPath.ToString();
```

Der Registry-Schlüssel:

```
HKEY_CURRENT_USER\Software\WindowsFormsApplication1\WindowsFormsApplication1\1.0.0.0
```

Das Verzeichnis:

```
C:\Users\Tom\AppData\Roaming\WindowsFormsApplication1\WindowsFormsApplication1\1.0.0.0
```

30.2.2 Methoden

Wichtige Methoden von *Application:*

Methode	Beschreibung
DoEvents	... ermöglicht Verarbeitung ausstehender Windows-Botschaften
EnableVisualStyles	... ermöglicht die Darstellung mit Styles
Exit	... Anwendung beenden

Methode	Beschreibung
ExitThread	... schließt aktuellen Thread
Run	... startet das Hauptfenster der Anwendung (Beginn einer Nachrichtenschleife für den aktuellen Thread)
Restart	... die Anwendung wird beendet und sofort neu gestartet
SetSuspendState	... versucht das System in den Standbymodus bzw. den Ruhezustand zu versetzen

Den Einsatz der *Run*-Methode haben wir ab Seite 1492 besprochen. Wie Sie *DoEvents* richtig verwenden, zeigen wir Ihnen im folgenden Beispiel.

BEISPIEL 30.17: Eine zeitaufwändige Berechnungsschleife blockiert die Anzeige der Uhrzeit. Durch Aufrufen von *DoEvents* läuft die Uhr auch während der Berechnung weiter.

C#

Zunächst wird die "Uhr" programmiert (vorher haben Sie eine *Timer*-Komponente in das Komponentenfach gezogen: *Interval = 1000*, *Enabled = True*):

```
private void timer1_Tick(object sender, EventArgs e)
{
    label1.Text = DateTime.Now.ToLongTimeString();
}
```

Die Berechnungsschleife wird gestartet:

```
private void button1_Click(object sender, EventArgs e)
{
    double a = 0;
    for (int i = 0; i < 100000000; i++)
    {
        a = Math.Sin(i) + a;
```

Nach jedem 1000ten Schleifendurchlauf wird CPU-Zeit freigegeben, sodass auf andere Ereignisse reagiert werden kann:

```
        if (i % 1000 ==0)  Application.DoEvents();
```

Wenn Sie obige Anweisung auskommentieren, "steht" die Uhr für die Zeit der Berechnung.

```
    }
```

Die Ergebnisanzeige:

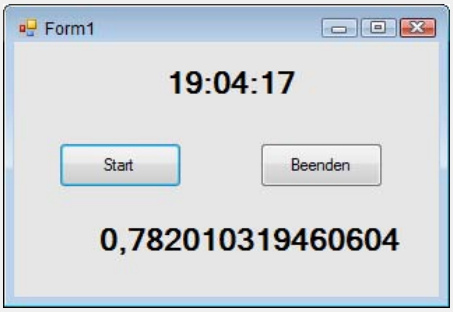
```
    label2.Text = a.ToString();
}
```

Anwendung beenden:

```
private void button2_Click(object sender, EventArgs e)
{
    Application.Exit();
}
}
```

BEISPIEL 30.17: Eine zeitaufwändige Berechnungsschleife blockiert die Anzeige der Uhrzeit. Durch Aufrufen von *DoEvents* läuft die Uhr auch während der Berechnung weiter.

Ergebnis



Eine recht effiziente Art, für Ruhe unter dem Schreibtisch zu sorgen, zeigt das folgende kurze Beispiel.

BEISPIEL 30.18: Den PC in den Ruhezustand versetzen

C#

```
...
    Application.SetSuspendState(PowerState.Suspend,false,true);
...
```

30.2.3 Ereignisse

Last but not least verfügt die *Application*-Klasse auch über diverse Ereignisse:

Ereignis	Beschreibung
ApplicationExit	... wenn die Anwendung beendet werden soll.
EnterThreadModal	... vor dem Eintritt der Anwendung in den modalen Zustand.
Idle	... die Applikation hat gerade nichts zu tun.
LeaveThreadModal	... der modale Zustand wird verlassen.
ThreadException	... eine nicht abgefangene Threadausnahme ist aufgetreten.
ThreadExit	... ein Thread wird beendet. Handelt es sich um den Hauptthread der Anwendung, wird zunächst dieses Ereignis und anschließend *Application-Exit* ausgelöst.

30.3 Allgemeine Eigenschaften von Komponenten

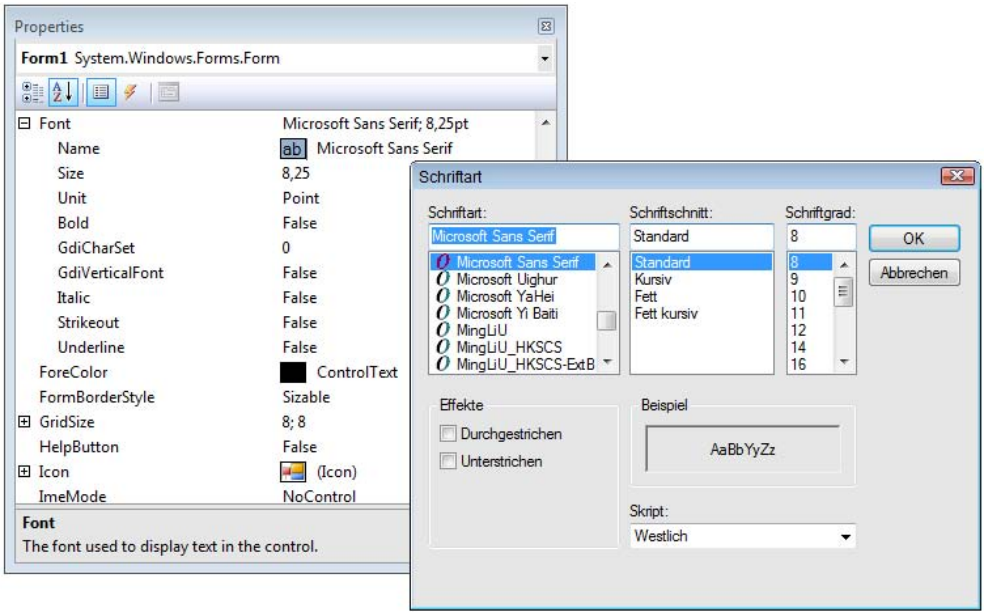
Es gibt eine Vielzahl von Standard-Properties über die (fast) alle Komponenten verfügen. Auf viele Eigenschaften kann nur zur Entwurfszeit, auf andere erst zur Laufzeit zugegriffen werden. Das ist auch der Grund, warum Letztere nicht im Eigenschaften-Fenster der Entwicklungsumgebung zu finden sind. Die Eigenschaften können zur Entwurfszeit (E) und/oder zur Laufzeit (L) verfügbar sein (r = nur lesbar).

Eigenschaft	Erläuterung	E	L
Anchor	Ausrichtung bezüglich des umgebenden Objekts	x	x
BackColor	Hintergrundfarbe	x	x
BorderStyle	Art des Rahmens	x	r
Cursor	Art des Cursors über dem Steuerelement	x	x
CausesValidation	siehe *Validate*-Ereignis		
ContextMenu	Auswahl eines Kontextmenüs	x	x
Dock	Andockstellen bezüglich des umgebenden Objekts	x	x
Enabled	aktiv/nicht aktiv	x	x
Font	Schriftattribute (Namen, Größe, fett, kursiv etc.)	x	x
ForeColor	Vordergrundfarbe, Zeichenfarbe	x	x
Handle	Fensterhandle, an welches das Control gebunden ist		r
Location	Position der linken oberen Ecke	x	x
Locked	Sperren des Steuerelements gegen Veränderungen	x	
Modifiers	Zugriffsmodifizierer (*private*, *public*, ...)	x	x
Name	Bezeichner	x	
TabIndex	Tab-Reihenfolge	x	x
TabStop	Tabulatorstopp Ja/Nein	x	x
Tag	Hilfseigenschaft (speichert Info-Text)	x	x
Text	Beschriftung oder Inhalt	x	x
Visible	Sichtbar Ja/Nein	x	x

Einige dieser Properties bedürfen einer speziellen Erläuterung.

30.3.1 Font

Mit dieser Eigenschaft (die in Wirklichkeit ein Objekt ist) werden Schriftart, Schriftgröße und Schriftstil der *Text*-Eigenschaft eingestellt (gilt nicht für die Titelleiste eines Formulars). Die Einstellung zur Entwurfszeit ist kein Problem, da ja zusätzlich auch ein komfortabler Fontdialog (der von Windows) zur Verfügung steht.



Der Zugriff auf die *Font*-Eigenschaften per Programmcode ist aber nicht mehr ganz so einfach.

BEISPIEL 30.19: Zugriff auf die *Font*-Eigenschaften

C#

Sie können zwar die Schriftgröße einer *TextBox* lesen:

```
MessageBox.Show(textBox1.Font.Size.ToString();    // zeigt z.B. 8,25
```

Ein direktes Zuweisen/Ändern der Schriftgröße ist allerdings nicht möglich, da diese Eigenschaft schreibgeschützt ist:

```
textBox1.Font.Size = 12;                    // Fehler (read only)
```

oder

```
textBox1.Font.Size = textBox2.Font.Size;    // Fehler (read only)
```

Um die Schrifteigenschaften zu ändern müssen Sie ein neues *Font*-Objekt erzeugen.

BEISPIEL 30.20: Ändern der Schriftgröße einer *TextBox*

C#

```
textBox1.Font = new Font(textBox1.Font.Name, 12);
```

Obiges Beispiel zeigt allerdings nur einen der zahlreichen Konstruktoren der *Font*-Klasse. Darunter gibt es natürlich auch welche, die das Zuweisen des Schriftstils (fett, kursiv, ...) ermöglichen.

Die möglichen Schriftstile sind in der *FontStyle*-Enumeration definiert (siehe folgende Tabelle):

Schriftstil	Beschreibung
Regular	normaler Text
Bold	fett formatierter Text
Italic	kursiv formatierter Text
Underline	unterstrichener Text
Strikeout	durchgestrichener Text

BEISPIEL 30.21: Inhalt einer *TextBox* fett formatieren

C#
```
textBox1.Font = new Font(textBox1.Font, FontStyle.Bold);
```

Was aber ist, wenn ein Text gleichzeitig mehrere Schriftstile haben soll? In diesem Fall müssen die einzelnen Schriftstile mit einer bitweisen ODER-Verknüpfung kombiniert werden.

BEISPIEL 30.22: Mehrere Schriftstile zuweisen

C#

Der Inhalt einer *TextBox* wird mit der Schriftart "Arial" und einer Schriftgröße von 16pt gleichzeitig fett, kursiv und mit Unterstreichung formatiert.

```
textBox1.Font = new Font("Arial", 16f, FontStyle.Bold | FontStyle.Underline |
                                       FontStyle.Italic);
```

Ergebnis

HINWEIS: Mehr zum *Font*-Objekt im Zusammenhang mit der Textausgabe auf einem Formular finden Sie im Grafik-Kapitel 33.

30.3.2 Handle

Die *Handle*-Eigenschaft ist wohl mehr für den fortgeschrittenen Programmierer bestimmt. Der Wert ist ein Windows-Handle und natürlich schreibgeschützt. Wenn das Handle noch nicht vorhanden ist, wird das Erstellen durch den Verweis auf diese Eigenschaft erzwungen.

BEISPIEL 30.23: Verwendung *Handle*

C#

Der Klick auf *button1* bewirkt, dass unter Verwendung des Handles von *button2* auch dieser mit einem Rechteck umrandet wird:

```
private void button1_Click(object sender, EventArgs e)
{
    ControlPaint.DrawFocusRectangle(Graphics.FromHwnd(button2.Handle),
                                    button2.ClientRectangle);
}
```

30.3.3 Tag

Diese Eigenschaft müssen Sie sich wie einen unsichtbaren "Merkzettel" vorstellen, den man an ein Objekt "anheften" kann. Im Eigenschaftenfenster erscheint *Tag* zwar als *string* Datentyp, in Wirklichkeit ist *Tag* aber vom *object*-Datentyp, denn per Programmcode können Sie dieser Eigenschaft einen beliebigen Wert zuweisen.

BEISPIEL 30.24: Verwendung *Tag*

C#

Sie haben der *Image*-Eigenschaft einer *PictureBox* ein Bild zugewiesen und wollen beim Klick auf die *PictureBox* einen erklärenden Text in einem Meldungsfenster anzeigen lassen:

```
...
pictureBox1.Tag = "Das ist meine Katze!";
...
private void pictureBox1_Click(object sender, EventArgs e)
{
    MessageBox.Show(pictureBox1.Tag.ToString());     // zeigt "Das ist meine Katze!"
}
```

HINWEIS: Seit .NET 2.0 verfügen auch Steuerelemente, die im Komponentenfach abgelegt werden, wie *Timer*, *OpenFileDialog*, ..., über eine *Tag*-Eigenschaft.

30.3.4 Modifiers

Beim visuellen Entwurfsprozess werden Steuerelemente vom Designer wie private Felder der *Form*-Klasse angelegt, ein Zugriff von außerhalb ist also zunächst nicht möglich.

Modifiers ist keine Eigenschaft im eigentlichen Sinn, sondern lediglich eine Option der Entwicklungsumgebung die Sie dann brauchen, wenn Sie von einem Formular aus auf ein Steuerelement zugreifen wollen, welches sich auf einem anderen Formular befindet. Angeboten werden die Modifizierer *public*, *protected*, *internal*, *protected internal* und *private*.

BEISPIEL 30.25: Verwendung von *Modifiers*

C#

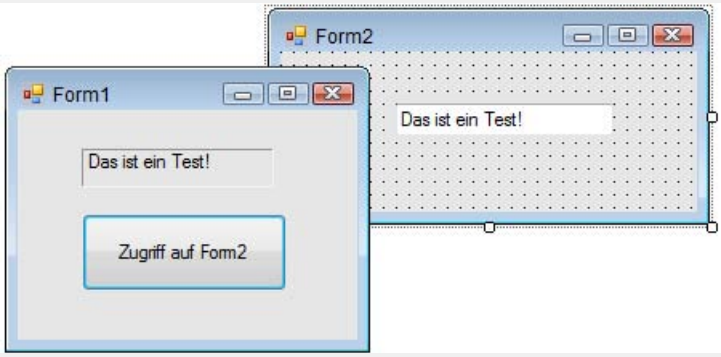
Nach dem Klick auf einen auf *Form1* befindlichen *Button* soll der Inhalt einer auf *Form2* befindlichen *TextBox* in einem *Label* auf *Form1* angezeigt werden. Der folgende Code:

```
public partial class Form1 : Form
{
 ...
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        // f2.Show();                    // nicht notwendig!
        label1.Text = f2.textBox1.Text;
    }
}
```

BEISPIEL 30.25: Verwendung von *Modifiers*

... funktioniert nur dann, wenn Sie vorher in *Form2* die *Modifiers*-Eigenschaft von *textBox1* auf *public, internal* oder *protected internal* gesetzt haben.

Ergebnis



30.4 Allgemeine Ereignisse von Komponenten

Wir wollen uns zunächst nur auf die Ereignisse beschränken, die für die meisten Komponenten gleichermaßen zutreffen, und auf einige grundsätzliche Programmieransätze eingehen.

HINWEIS: Wer sich fundiert mit dem Microsoft Ereignismodell auseinandersetzen möchte sei auf das Kapitel 26 (Microsoft Event Pattern) verwiesen!

30.4.1 Die Eventhandler-Argumente

Jedem Ereignis-Handler werden zumindest zwei Parameter übergeben:

- das *sender*-Objekt,
- die eigentlichen Parameter als Objekt mit dem kurzen Namen ***e***.

Was können wir mit beiden Werten anfangen?

30.4.2 Sender

Da Ereignis-Handler einen beliebigen Namen erhalten und auch "artfremde" Komponenten sich ein und denselben Eventhandler teilen können, muss es ein Unterscheidungsmerkmal für den Aufrufer des Ereignisses geben. Über den *sender* ist es möglich zu entscheiden, welches Objekt das Ereignis ausgelöst hat.

Zunächst werden Sie sicher enttäuscht sein, wenn Sie sich die einzelnen Eigenschaften bzw. Methoden des *sender*-Objekts ansehen, lediglich eine Methode *GetType* ist zu finden. Doch keine Sorge, typisieren Sie das Objekt, so haben Sie Zugriff auf alle objekttypischen Eigenschaften und Methoden der das Ereignis auslösenden Komponente.

BEISPIEL 30.26: Ein gemeinsamer Ereignis-Handler (*TextChanged*) für mehrere Textboxen zeigt in der Kopfzeile des Formulars den Inhalt der jeweils bearbeiteten Textbox an.

C#
```
private void textBox_TextChanged(object sender, EventArgs e)
{
    this.Text = (sender as TextBox).Text;
}
```

Im obigen Beispiel wurde der *as*-Operator zur Typkonvertierung eingesetzt. Aber auch der übliche explizite *()*-Operator ist anwendbar[1].

BEISPIEL 30.27: Vorgängerbeispiel mit dem *()*-Konvertierungsoperator

C#
```
private void textBox_TextChanged(object sender, EventArgs e)
{
    this.Text = ((TextBox) sender).Text;
}
```

Im Folgenden werden wir aber den *as*-Operator bevorzugen, da dieser *null* liefert, falls die Konvertierung misslingt. Beim *()*-Konvertierungsoperator würde hingegen eine Exception ausgelöst.

Das wäre zum Beispiel der Fall, wenn ein *TextChange* von einer *TextBox* und einer *ComboBox* kommt. Spätestens beim Testen der Anwendung gäbe es Ärger, da der Typ nicht immer stimmt. Mit dem *as*-Operator hingegen ist eine vorangestellte *null*-Abfrage möglich.

BEISPIEL 30.28: Vorgängerbeispiel mit *as*-Konvertierungsoperator und *null*-Abfrage

C#
```
private void control_TextChanged(object sender, EventArgs e)
{
    if (sender as TextBox != null) this.Text = (sender as TextBox).Text;
}
```

Brauchen Sie unbedingt den speziellen Objekttyp, können Sie zunächst mittels *GetType()* den Typ von *sender* feststellen um dann, in Abhängigkeit vom Typ, unterschiedliche Aktionen auszuführen.

BEISPIEL 30.29: Verwendung von *sender*

C#

Auch den folgenden Eventhandler teilen sich mehrere *Text*- und *ComboBox*en gemeinsam. Ändert sich der Inhalt einer *TextBox*, so wird dieser in der Titelleiste des Formulars angezeigt. Ändert sich aber die *Text*-Eigenschaft einer *ComboBox*, so wird nur deren Namen angezeigt.

```
private void control_TextChanged(object sender, EventArgs e)
{
    switch (sender.GetType().Name)
    {
        case "TextBox": this.Text = (sender as TextBox).Text; break;
        case "ComboBox": this.Text = (sender as ComboBox).Name; break;
```

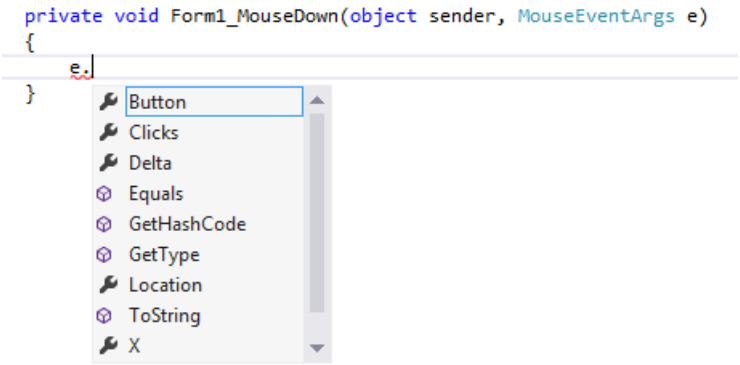
[1] Im Unterschied zum *()*-Konvertierungsoperator ist der *as*-Operator nur für Referenz- bzw. Verweistypen anwendbar, wozu auch alle Controls gehören.

BEISPIEL 30.29: Verwendung von *sender*

```
    }
}
```

30.4.3 Der Parameter e

Was sich hinter dem Parameter *e* versteckt, ist vom jeweiligen Ereignis abhängig. So werden bei einem *MouseDown*-Ereignis unter anderem die gedrückten Tasten und die Koordinaten des Mausklicks im Parameter *e* übergeben:



Teilweise werden über diesen Parameter auch Werte an das aufrufende Programm zurückgegeben.

BEISPIEL 30.30: Das Schließen des Formulars wird verhindert.

C#
```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = true;
}
```

30.4.4 Mausereignisse

Wenn sich der Mauskursor über einem Objekt befindet, können die folgenden Mausaktivitäten (teilweise mit Übergabeparametern) ausgewertet werden:

Ereignis	... tritt ein, wenn
Click	… auf das Objekt geklickt wird.
DblClick	… auf das Objekt doppelt geklickt wird.
MouseDown	… eine Maustaste niedergedrückt wird.
MouseUp	… eine Maustaste losgelassen wird.
MouseMove	… die Maus bewegt wird.

Ereignis	... tritt ein, wenn
MouseEnter	… wenn die Maus in das Control hinein bewegt wird.
MouseLeave	… wenn die Maus aus dem Control hinaus bewegt wird.

Bei einem *MouseDown* können Sie über *e.Button* unterscheiden, welcher Button gerade gedrückt wurde (*Left, Middle, Right*). Gleichzeitig können Sie über *e.x* bzw. *e.y* die Koordinaten bezüglich des jeweiligen Objekts ermitteln.

HINWEIS: Beachten Sie, dass jeder Doppelklick auch ein "normales" Klickereignis auslöst. Man sollte deshalb überlegt zu Werke gehen, wenn für ein Control beide Events gleichzeitig besetzt werden sollen.

BEISPIEL 30.31: *MouseDown*

C#

Beim Niederdrücken der rechten Maustaste über dem Formular wird eine *MessageBox* erzeugt, wenn sich die Maus in dem durch die Koordinaten 10,10 (linke obere Ecke) und 110,110 (rechte untere Ecke) bezeichneten Rechteck befindet:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    if ((e.Button == MouseButtons.Right) &
            new Rectangle(10, 10, 100, 100).Contains(e.X, e.Y)) MessageBox.Show("Erfolg");
}
```

BEISPIEL 30.32: Ändern der *ListBox*-Hintergrundfarbe, wenn die Maus darüber bewegt wird.

C#

```
private void listBox1_MouseEnter(object sender, EventArgs e)
{
    (sender as ListBox).BackColor = Color.Blue;
}

private void listBox1_MouseLeave(object sender, EventArgs e)
{
    (sender as ListBox).BackColor = Color.White;
}
```

Tastaturereignisse

Wenn ein Steuerelement den Fokus hat, können für dieses Objekt die in der folgenden Tabelle aufgelisteten Keyboard-Events ausgewertet werden.

Ereignis	... tritt ein, wenn
KeyPress	... eine Taste gedrückt wird.
KeyDown	... die Taste nach unten bewegt wird (mit Intervall).
KeyUp	... eine Taste losgelassen wird.

KeyPress registriert das Zeichen der gedrückten Taste, während *KeyDown* und *KeyUp* auf alle Tasten der Tastatur (einschließlich Funktionstasten und Tastenkombinationen mit den Tasten *Umschalt*, *Alt* und *Strg*) reagieren können.

BEISPIEL 30.33: Verwendung *KeyUp*

C#

Beim Loslassen einer Zifferntaste innerhalb einer *TextBox* wird die Ziffer in eine *ListBox* übernommen (48 ... 57 sind die ANSI-Codes der Ziffern 0 ... 9).

```
private void textBox1_KeyUp(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
                listBox1.Items.Add((e.KeyValue-48).ToString());
}
```

30.4.5 KeyPreview

KeyPreview ist kein Ereignis, sondern eine Formulareigenschaft! *KeyPreview* steht aber im engen Zusammenhang mit den Tastaturereignissen und soll deshalb (ausnahmsweise) bereits an dieser Stelle erwähnt werden. Wenn *KeyPreview* den Wert *False* hat (Voreinstellung), werden die Ereignisse sofort zur Komponente weitergegeben. Hat *KeyPreview* aber den Wert *True*, so gehen, unabhängig von der aktiven Komponente, die Tastaturereignisse *KeyDown*, *KeyUp* und *KeyPress* zuerst an das Formular. Erst danach wird das Tastaturereignis an das Steuerelement weitergereicht. Damit kann an zentraler Stelle auf Tastaturereignisse reagiert werden.

BEISPIEL 30.34: *KeyPreview*

C#

Entsprechend dem vorhergehenden Beispiel soll beim Drücken einer Zifferntaste diese in eine *ListBox* übernommen werden, in diesem Fall jedoch bei **allen** Controls des aktuellen Fensters.

Setzen Sie im Eigenschaftenfenster (F4) die Eigenschaft *KeyPreview* des Formulars auf *True* und erzeugen Sie folgenden Event-Handler:

```
private void Form1_KeyUp(object sender, KeyEventArgs e)
{
    if ((e.KeyValue > 47) & (e.KeyValue < 58))
        listBox1.Items.Add((e.KeyValue - 48).ToString());
    e.Handled = true;
}
```

HINWEIS: Da in diesem Beispiel bei *KeyPreview=True* das Ereignis nur im Event-Handler des Formulars und nicht auch im aktuellen Steuerelement verarbeitet werden soll, haben wir *e.Handled* auf *True* gesetzt.

30.4.6 Weitere Ereignisse

Die folgenden Events finden Sie ebenfalls bei einer Vielzahl von Objekten:

Ereignis	... tritt ein, wenn
Change	… der Inhalt der Komponente geändert wird.
Enter	… die Komponente den Fokus erhält.
DragDrop	… das Objekt über der Komponente abgelegt wird.
DragOver	… das Objekt über die Komponente gezogen wird.
HelpRequested	… die Hilfe angefordert wird (F1).
Leave	… die Komponente den Fokus verliert.
Paint	… das Steuerelement gezeichnet wird.
Resize	… die Komponente in der Größe verändert wird.
Validate	… der Inhalt von Steuerelementen überprüft wird.

BEISPIEL 30.35: Erhält die *TextBox* den Eingabefokus, soll sich die Hintergrundfarbe ändern.

C#

```
private void textBox1_Enter(object sender, EventArgs e)
{
    textBox1.BackColor = Color.Yellow;
}
```

Beim Verlassen stellen wir die normale Farbe ein:

```
private void textBox1_Leave(object sender, EventArgs e)
{
    textBox1.BackColor = Color.White;
}
```

Im Zusammenhang mit dem Auftreten der Fokus-Ereignisse spielt häufig auch die Reihenfolge eine Rolle:

Enter ⇨ *GotFocus* ⇨ *Leave* ⇨ *Validating* ⇨ *Validated* ⇨ *LostFocus*

30.4.7 Validätsprüfungen

Das *Validate*-Ereignis ermöglicht es, zusammen mit der *CausesValidation*-Eigenschaft den Inhalt von Steuerelementen zu prüfen, **bevor** der Fokus das Steuerelement verlässt. Das *Validate*-Ereignis eignet sich besser zum Überprüfen der Dateneingabe als das *LostFocus*-Ereignis, da *LostFocus* erst **nach** dem Verschieben des Fokus eintritt.

Validate wird nur dann ausgelöst, wenn der Fokus in ein Steuerelement wechselt, bei dem die *CausesValidation*-Eigenschaft *True* ist (Standardeinstellung). *CausesValidation* sollte nur bei den Controls auf *False* gesetzt werden, deren Aktivierung keine Validätskontrolle auslösen soll, wie z.B. eine *Abbrechen*- oder eine *Hilfe*-Schaltfläche.

HINWEIS: Ist die Prüfung innerhalb des *Validating*-Events nicht erfolgreich, können Sie mit *e.Cancel* die weitere Ereigniskette (siehe oben) abbrechen.

BEISPIEL 30.36: *Validating*

C#

Der Fokus wandert nur dann zum nächsten Steuerelement, wenn in die *TextBox* mehr als fünf Zeichen eingegeben werden.

```
private void textBox2_Validating(object sender, CancelEventArgs e)
{
    if (textBox2.Text.Length < 5)
    {
        MessageBox.Show("Bitte mehr als 5 Zeichen eingeben!");
        e.Cancel = true;
    }
}
```

30.4.8 SendKeys

Mit diesem Objekt werden Tastatureingaben durch den Bediener simuliert, daher ist es zweckmäßig, es bereits an dieser Stelle im Zusammenhang mit Tastaturereignissen zu erwähnen. Zwei Methoden stehen zur Auswahl:

- *Send*
- *SendWait*

Während Erstere sich damit begnügt, die Tastatureingaben einfach an die aktive Anwendung zu senden, wartet *SendWait* auch darauf, dass die Daten verarbeitet werden. Insbesondere bei etwas langsameren Operationen kann es sonst schnell zu einem Fehlverhalten kommen.

Das Argument der beiden Methoden ist eine Zeichenkette. Jede Taste wird dabei durch mindestens ein Zeichen repräsentiert.

HINWEIS: Das Pluszeichen (+), Caret-Zeichen (^) und Prozentzeichen (%) sind für die UMSCHALT-, STRG- und ALT-Taste vorgesehen. Sondertasten sind in geschweifte Klammern einzuschließen.

BEISPIEL 30.37: *SendKeys*

C#

Die folgende Anweisung sendet die Tastenfolge *Alt+F4* an das aktive Fenster und bewirkt damit ein Schließen der Applikation.

```
private void button1_Click(object sender, EventArgs e)
{
    SendKeys.Send('%{F4}');
}
```

Häufig soll sich die "Tastatureingabe" nicht auf das aktuelle Formular, sondern auf das aktive Steuerelement beziehen. Dann muss dieses Steuerelement vorher den Fokus erhalten.

BEISPIEL 30.38: *SendWait* mit *SetFocus*

C#

Die folgende Sequenz füllt das Textfeld *TextBox1* mit den Ziffern 12345678 und setzt danach die Ausführung fort.

```
private void button1_Click(object sender, EventArgs e)
{
    textBox1.Focus();
    SendKeys.SendWait('12345678');
}
```

HINWEIS: *SendKeys* macht es auch möglich, quasi "wie von Geisterhand" andere Windows-Programme (z.B. den integrierten Taschenrechner) aufzurufen. Mehr dazu finden Sie in unserem [Visual C# 2012 Kochbuch].

30.5 Allgemeine Methoden von Komponenten

Auch hier nur ein Auszug aus dem reichhaltigen Sortiment:

Methode	Erläuterung
Contains	… kontrolliert, ob ein angegebenes Steuerelement dem aktuellen untergeordnet ist.
CreateGraphics	… erzeugt ein *Graphics*-Objekt zum Zeichnen im Steuerelement (mehr dazu in Kapitel 33).
DoDragDrop	… beginnt eine Drag&Drop-Operation.
FindForm	… ruft das übergeordnete Formular ab.
Focus	… setzt den Fokus auf das Objekt.
GetContainerControl	… ruft das übergeordnete Steuerelement ab.
GetType	… ruft den Typ des Steuerelements ab.
Hide	… verbirgt das Steuerelement.
Invalidate	… veranlasst das Neuzeichnen des Controls.
Refresh	… erneuert den Aufbau des Steuerelements.
Scale	… skaliert das Steuerelement.
SelectNextControl	… aktiviert das folgende Steuerelement.
SetBounds	… setzt die Größe des Steuerelements.
Show	… zeigt das Steuerelement nach dem Verbergen wieder an.

HINWEIS: Über die an die Methoden zu übergebenden Parameter informieren Sie sich am besten per Intellisense bzw. in der Dokumentation.

BEISPIEL 30.39: *SetBounds*

C#

Die folgende Anweisung verschiebt das aktuelle Formular an die Position 10,10 und verändert gleichzeitig die Größe auf 100 x 100 Pixel.

```
this.SetBounds(10, 10, 100, 100);
```

BEISPIEL 30.40: Skalieren einer *GroupBox* mit dem Faktor 1,5

C#

```
groupBox1.Scale(1.5, 1.5);
```

Kapitel 31

Windows Forms-Formulare

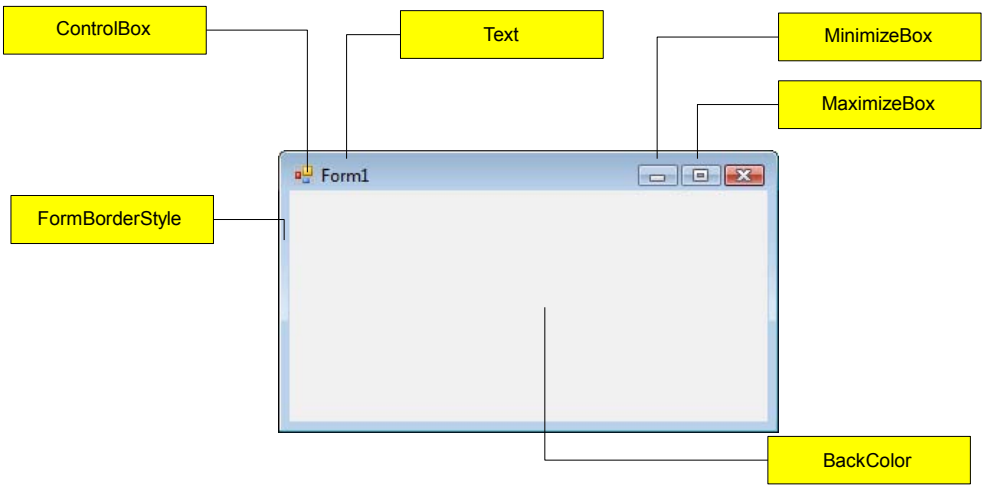
Fast jede Windows-Applikation läuft in einem oder mehreren Fenstern ab, die sozusagen als Container für die Anwendung fungieren. Schon deshalb ist das Formular (*Form*) das wichtigste Objekt, wir werden uns also damit etwas ausführlicher auseinander setzen müssen.

HINWEIS: Die Ausführungen zu den allgemeinen Eigenschaften/Methoden und Ereignissen aus dem vorhergehenden Kapitel treffen auch auf die Windows Formulare zu. Sie sollten also das Kapitel 30 bereits durchgearbeitet haben.

31.1 Übersicht

Haben Sie ein neues Windows Forms-Anwendungsprojekt geöffnet, werden Sie bereits von einem ersten Formular im Designer begrüßt.

Wie ein Formular aussieht, brauchen wir sicher nicht weiter zu erklären, welche Eigenschaften jedoch für das Aussehen verantwortlich sind, soll die folgende Abbildung verdeutlichen:



Für die weitere Orientierung sollen die folgenden Tabellen sorgen, die in Kürze die wichtigsten Eigenschaften, Methoden und Ereignisse des Formulars erläutern. In den weiteren Abschnitten wenden wir uns dann spezifischen Themen rund um das Formular zu.

31.1.1 Wichtige Eigenschaften des Form-Objekts

Eigenschaft	Beschreibung
AcceptButton *CancelButton*	Diese Eigenschaften legen fest, welche Buttons mit den Standardereignissen *Enter*-Taste bzw. *Esc*-Taste verknüpft werden. Diese Funktionalität wird häufig in Dialogboxen verwendet (siehe auch *DialogResult*). Löst der Nutzer eine der beiden Aktionen aus, wird der Ereigniscode der jeweiligen Taste verarbeitet.
ActiveControl	… bestimmt das gerade aktive Control, z.B: `if (sender == this.ActiveControl) berechne();`
ActiveForm	… bestimmt das aktive Formular der Anwendung.
ActiveMdiChild	… ermittelt das gerade aktive MDI-Child-Fenster.
AutoScroll	… bestimmt, ob das Formular automatisch Scrollbars einfügen soll, wenn der Clientbereich nicht komplett darstellbar ist.
BackgroundImage	… eine Grafik für den Hintergrund.
ClientSize	… ermittelt die Größe des Formular-Clientbereichs.
ContextMenu	… das Kontextmenü des Formulars.
ControlBox	… Anzeige des Systemmenüs (*True/False*).
Controls	… eine Collection aller enthaltenen Controls.
Cursor	… legt die Cursorform für das aktuelle Formular fest.
DesktopBounds	… legt Position und Größe des Formulars auf dem Desktop fest, z.B. `this.DesktopBounds = (new Rectangle.Create(10, 10, 100, 100));`
DesktopLocation	… legt die Position des Formulars fest.

DialogResult	… über diesen Wert können Dialog-Statusinformationen an ein aufrufendes Programm zurückgegeben werden, z.B.
	`Form2 f2 = new Form2();`
	`if (f2.ShowDialog() == DialoResult.Abort) { ...}`

DoubleBuffered	… ermöglicht flackerfreie Darstellung von Grafiken (siehe Kapitel 36)
Dock	… setzt die Ausrichtung gegenüber einem übergeordneten Fenster.
DockPadding	… setzt den Zwischenraum beim Docking von Controls.
FormBorderStyle	… setzt den Formularrahmen. Dieser hat auch Einfluss auf das Verhalten (Tool-Window, Dialog etc.).
HelpButton	… soll der Hilfe-Button angezeigt werden?
Icon	… das Formular-Icon.

Eigenschaft	Beschreibung
IsMdiChild	… handelt es sich um ein MDI-Child-Fenster?
IsMdiContainer	… handelt es sich um ein MDI-Container-Fenster?
Location	… die linke obere Ecke des Formulars.
MaximizeBox *MinimizeBox*	… Anzeige der beiden Formular-Buttons (*True/False*).
MaximumSize *MinimumSize*	… setzt maximale bzw. minimale Maße für das Fenster.
MdiChildren	… eine Collection der untergeordneten MDI-Child-Fenster.
MdiParent	… ermittelt den MDI-Container.
Menu	… das Hauptmenü des Formulars.
Modal	… wird das Formular (Dialog) modal angezeigt?
Opacity	… Transparenz des Formulars in Prozent.
OwnedForms	… eine Collection der untergeordneten Formulare.
Owner	… das übergeordnete Formular.
ShowInTaskbar	… soll das Formular in der Taskbar angezeigt werden?
Size	… die Formulargröße.
StartPosition	… wo wird das Fenster beim ersten Aufruf angezeigt (zentriert etc.)?
Text	… der Text in der Titelleiste.
TopMost	… soll das Formular an oberster Position angezeigt werden?
TransparencyKey	… welche Formularfarbe soll transparent dargestellt werden?
Visible	… ist das Formular sichtbar?
WindowState	… ist das Fenster maximiert, minimiert oder normal dargestellt?

Controls-Auflistung

Mit dieser Eigenschaft ist der Zugriff auf alle im Formular vorhandenen Steuerelemente möglich.

BEISPIEL 31.1: Alle Controls im aktuellen Formular um zehn Pixel nach links verschieben

C#
```
foreach (Control c in this.Controls)
{
    c.Left -= 10;
}
```

Der Zugriff auf Elemente der *Controls*-Auflistung eines Formulars ist nicht nur über den Index, sondern auch über den Namen des Elements möglich.

BEISPIEL 31.2: Zugriff auf ein spezielles Control über dessen Namen

C#
```
TextBox tb = (TextBox) Controls["textBox2"];
MessageBox.Show(tb.Text);
```

31.1.2 Wichtige Ereignisse des Form-Objekts

Neben den bereits im vorhergehenden Kapitel aufgelisteten Events sind für ein Formular die folgenden Events von Bedeutung:

Ereignis	... tritt ein, wenn
Activated	... das Formular aktiviert wird.
FormClosing	… das Formular geschlossen werden soll. Sie können den Vorgang über den Parameter *e.Cancel* abbrechen und so ein Schließen des Formulars verhindern.
FormClosed	… das Formular geschlossen ist.
Deactivate	… ein anderes Formular aktiviert wird.
Load	… das Formular geladen wird.
Paint	… das Formular neu gezeichnet werden muss.
Resize	… die Größe eines Formulars verändert wird.
HelpRequested	… Hilfe vom Nutzer angefordert wird (F1).
Layout	… die untergeordneten Controls neu positioniert werden müssen (Größenänderung des Formulars oder Hinzufügen von Steuerelementen).
LocationChanged	… sich die Position des Formulars ändert.
MdiChildActivated	… wenn ein MDI-Child-Fenster aktiviert wird.

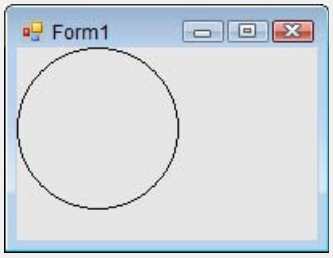
BEISPIEL 31.3: Verwenden des *Paint*-Events zum Zeichnen eines Kreises

C#

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Ergebnis

Mit jedem Anzeigen oder jeder Größenänderung wird *Paint* ausgelöst. Das Ergebnis:



HINWEIS: Mehr zur Grafikprogrammierung finden Sie im Kapitel 33.

Beim Laden eines Formulars treten – nach Aufruf des Konstruktors – die Ereignisse in folgender Reihenfolge auf:

Move ⇨ *Load* ⇨ *Layout* ⇨ *Activated* ⇨ *Paint*

Beim Schließen hingegen haben wir es mit folgender Ereigniskette zu tun:

FormClosing ⇨ *FormClosed* ⇨ *Deactivate*

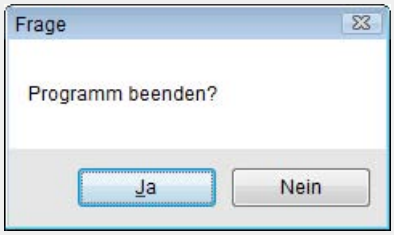
HINWEIS: Das Praxisbeispiel 31.4.2 zeigt, wie Sie obige Ereignisketten selbst experimentell ermitteln können!

BEISPIEL 31.4: Sicherheitsabfrage vor dem Schließen des Formulars

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = MessageBox.Show("Programm beenden?", "Frage", MessageBoxButtons.YesNo) ==
                                        DialogResult.No;
}
```

Ergebnis



31.1.3 Wichtige Methoden des Form-Objekts

Die wichtigsten Methoden für Formulare sind im Folgenden zusammengestellt.

Methode	Beschreibung
Activate	… aktiviert das Formular.
BringToFront	… verschiebt das Formular an die oberste Position (innerhalb der Anwendung).
Close	… schließt das Formular.
CreateControl	… erzeugt ein neues Steuerelement.
CreateGraphics	… erstellt ein *Graphics*-Objekt für die grafische Ausgabe.
DoDragDrop	… startet eine Drag & Drop-Operation.
Focus	… setzt den Fokus auf das Formular.
GetNextControl	… liefert das folgende Control in der Tab-Reihenfolge.
Hide	… verbirgt das Formular.
Invalidate	… erzwingt ein Neuzeichnen des Formularinhalts.

Methode	Beschreibung
PointToClient *RectangleToClient*	… rechnet Screen-Koordinaten in Fensterkoordinaten um (je nach Fensterposition).
PointToScreen *RectangleToScreen*	… rechnet Fensterkoordinaten in Screen-Koordinaten um.
Refresh	… erzwingt ein Neuzeichnen des Fensters und der untergeordneten Controls.
SelectNextControl	… verschiebt den Eingabefokus bei den untergeordneten Controls.
Show	… zeigt das Fenster an.
ShowDialog	… zeigt das Fenster als Dialogbox (modal) an.

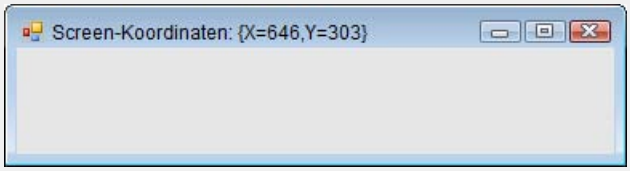
BEISPIEL 31.5: Umrechnen in Screen-Koordinaten

C#

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    this.Text = "Screen-Koordinaten: " + this.PointToScreen(new Point(e.X, e.Y)).ToString();
}
```

Ergebnis

Die Anzeige hängt jetzt nicht nur von der relativen Mausposition, sondern auch von der absoluten Position des Formulars ab:



HINWEIS: Statt des Namens des Formulars (z.B. *Form1*) können Sie auch *this* verwenden oder den Bezeichner völlig weglassen, da der Code ja in der aktuellen Klasse ausgeführt wird.

31.2 Praktische Aufgabenstellungen

Im folgenden wollen wir die o.g. Eigenschaften, Methoden und Ereignisse nutzen, um einige recht praktische Aufgabenstellungen im Umgang mit Windows-Formularen zu lösen.

31.2.1 Fenster anzeigen

Wie Sie aus der Startprozedur *Main()* heraus ein Fenster aufrufen, wurde im vorhergehenden Kapitel beschrieben. Wie Sie aus dem Hauptfenster heraus weitere Formulare aufrufen, soll Mittelpunkt dieses Abschnitts sein.

Zwei grundsätzliche Typen von Formularen müssen Sie unterscheiden:

- modale Fenster (Dialoge)
- nichtmodale Fenster

Die Unterscheidung zwischen beiden Varianten findet erst beim Aufruf bzw. bei der Anzeige eines *Form*-Objekts statt, zur Entwurfszeit wird diese Unterscheidung nicht getroffen, sieht man einmal von unterschiedlichen Rahmentypen (*FormBorderStyle*-Eigenschaft) ab.

Nichtmodale Fenster

Hierbei handelt es sich um ein Fenster, das den Fokus auch an andere Fenster abgeben bzw. das auch verdeckt werden kann.

Die Anzeige erfolgt mit Hilfe der Methode *Show*, die asynchron ausgeführt wird, d.h., es wird mit der Programmverarbeitung nicht auf das Schließen des Formulars gewartet.

HINWEIS: Vor der Anzeige des Formulars muss dieses mit *new* instanziiert werden!

BEISPIEL 31.6: Instanziieren und Anzeigen eines weiteren Formulars

C#
```
private void button1_Click(object sender, EventArgs e)
{
    Form2 f2 = new Form2();
    f2.Text = "Mein zweites Formular";
    f2.Show();
}
```

Der "Nachteil" dieser Fenster: Sie können zum einen verdeckt werden, zum anderen wissen Sie als Programmierer nie, wann der Anwender das Fenster schließt[1]. Für die Eingabe von Werten und deren spätere Verarbeitung sind sie also ungeeignet.

Modale Fenster (Dialoge)

Abhilfe schaffen die Dialogfenster, auch modale Fenster genannt. Diese werden statt mit *Show* mit der Methode *ShowDialog* angezeigt. Die Programmausführung wird mit dem Aufruf der Methode an dieser Stelle so lange gestoppt, bis der Nutzer das Formular wieder geschlossen hat.

BEISPIEL 31.7: Anzeige eines Dialogfensters

C#
```
    Form2 f2 = new Form2();
    f2.Text = "Bitte tragen Sie Ihren Namen ein ...";
    f2.ShowDialog();
    MessageBox.Show(f2.textBox1.Text);      // Anzeige des Eingabewerts
```

[1] Natürlich könnten Sie die entsprechenden Ereignisse auswerten, aber übersichtlicher wird Ihr Programm dadurch sicher nicht, vor allem dann nicht, wenn es um eine synchrone Programmabarbeitung geht.

HINWEIS: Bevor Sie auf Controls in *Form2* zugreifen können, müssen Sie deren *Modifiers*-Eigenschaft auf *public* festgelegt haben. Andernfalls können Sie nicht mit den Controls arbeiten.

Zu einer ordentlichen Dialogbox gehören im Allgemeinen auch ein OK- und ein Abbruch-Button. Auf diese Weise kann im aufrufenden Programm schnell entschieden werden, welcher Meinung der Anwender beim Schließen der Dialogbox war. Von zentraler Bedeutung ist in diesem Fall der Rückgabewert der Methode *ShowDialog*.

BEISPIEL 31.8: Auswerten des Rückgabewertes

C#

```
Form2 f2 = new Form2();
if (f2.ShowDialog() == DialogResult.Abort)
    {...}
```

Die möglichen Rückgabewerte:

DialogResult
Abort
Cancel
Ignore
No
None
OK
Retry
Yes

In der Dialogbox selbst stellen Sie den Rückgabewert entweder durch das direkte Setzen der Eigenschaft *DialogResult* ein oder Sie weisen den beiden Buttons (OK, Abbruch) die gewünschte *DialogResult*-Eigenschaft zu.

BEISPIEL 31.9: Verwendung von *DialogResult*

C#

Auf *Form1* befinden sich ein *Label* und ein *Button*. Über Letzteren wird ein Dialogfenster *Form2* aufgerufen, das die Schaltflächen "OK" und "Abbruch" besitzt. Außerdem hat *Form2* eine *TextBox*, deren *Modifiers*-Eigenschaft Sie auf *public* setzen.

```
public partial class Form1 : Form
{ ...
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.Text = "Bitte tragen Sie Ihren Namen ein ... ";
        if (f2.ShowDialog() == DialogResult.OK)
                        label1.Text = f2.textBox1.Text;
```

BEISPIEL 31.9: Verwendung von *DialogResult*

```
    }
}
```

Der Code von *Form2*:

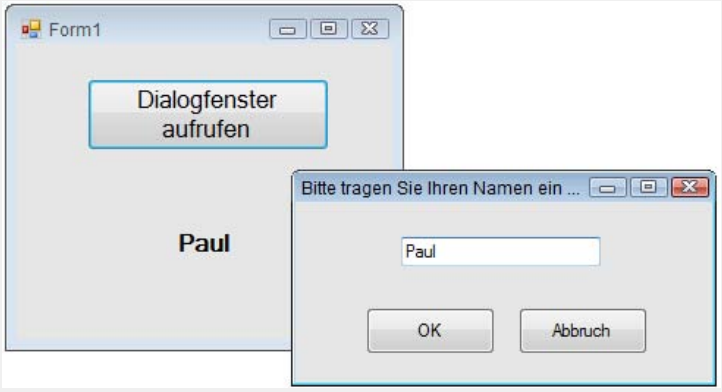
```
public partial class Form2 : Form
{
    public Form2()
    {
        InitializeComponent();
```

Die folgenden beiden Zuweisungen können Sie auch direkt im Eigenschaftenfenster von *button1* bzw. *button2* vornehmen:

```
        button1.DialogResult = DialogResult.OK;
        button2.DialogResult = DialogResult.Abort;
    }
}
```

Ergebnis

Nach dem Klick auf "OK" wird der in *Form1* eingetragene Wert in *Form2* angezeigt.



31.2.2 Splash Screens beim Anwendungsstart anzeigen

Von vielen kommerziellen Anwendungen ist Ihnen sicher die Funktion eines Splash Screens bekannt. Der Hintergrund ist in vielen Fällen, dass die Zeit für das Laden von Programmmodulen, Datenbanken etc. für den Endanwender irgendwie sinnvoll überbrückt werden soll, ohne dass der Verdacht aufkommt, die Anwendung "hängt". An einem recht einfachen Vertreter dieser Gattung wollen wir Ihnen die prinzipielle Vorgehensweise demonstrieren.

BEISPIEL 31.10: Einsatz eines Splash Screens

C#

Erstellen Sie zunächst ein neues Projekt und fügen Sie diesem neben dem Standardformular *Form1* ein weiteres Formular *SplashScreen* hinzu. Diesem gilt auch zunächst unsere Aufmerksamkeit.

Fügen Sie *SplashScreen* einen *Label* für die Begrüßungsmeldung und einen *Label* für mögliche Statusmeldungen hinzu. Setzen Sie weiterhin die *FormBorderStyle*-Eigenschaft auf den von Ihnen favorisierten Wert (z.B. *None*), sowie die Eigenschaft *StartPosition* auf *ScreenCenter*. Nicht vergessen dürfen wir auch die Eigenschaft *TopMost,* die wir tunlichst auf *true* setzen sollten, damit unser Formular auch im Vordergrund steht und nicht hinter dem Hauptformular der Anwendung verloren geht.

Die Klassendefinition erweitern Sie bitte um die im Folgenden fett hervorgehobenen Einträge:

```
...
    public partial class SplashScreen : Form
    {
```

Da wir nur ein Formular benötigen, definieren wir diese gleich intern und statisch:

```
        static SplashScreen frmSplashScreen = null;
```

Mit der folgenden *Start*-Methode wird das Formular initialisiert und angezeigt. Da die Methode statisch ist, können wir diese direkt mit *SplashScreen.Start()* aufrufen.

```
        static public void Start()
        {
            if (frmSplashScreen != null)
                return;
            frmSplashScreen = new SplashScreen();
            frmSplashScreen.Show();
        }
```

Auch das Schließen des Formulars übernimmt eine statische Methode der *SplashScreen*-Klasse:

```
        static public void Stop()
        {
            frmSplashScreen.Close();
            frmSplashScreen = null;
        }
```

Möchten Sie Statusmeldungen anzeigen, ist dies über die statische Methode *SetMessage* möglich. Die Meldungen werden im *label2* angezeigt:

```
        static public void SetMessage(string msg)
        {
           frmSplashScreen.label2.Text = msg;
           Application.DoEvents();
        }
...
```

BEISPIEL 31.10: Einsatz eines Splash Screens

Selbstverständlich können Sie auch noch Fortschrittsanzeigen etc. in diesem Formular unterbringen, das Grundprinzip dürfte jedoch weitgehend gleich bleiben.

Was fehlt ist die Integration des neuen Splash-Screens in Ihre Anwendung. Öffnen Sie dafür die Datei *Program.cs* und nehmen Sie folgende Erweiterung vor:

```
static class Program
{
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        SplashScreen.Start();
        Application.Run(new Form1());
    }
}
```

Last but not least, ist auch unser eigentliches Programm von Interesse. Wir wollen hektische und langwierige Datenbankoperationen beim Öffnen von *Form1* mit einer einfachen Schleife und der *Thead.Sleep*-Methode simulieren:

```
...
    public partial class Form1 : Form
    {
        private void Form1_Load(object sender, EventArgs e)
        {
            for (int i = 0; i < 100; i++)
            {
                Application.DoEvents();
                Thread.Sleep(50);
```

Hier besteht die Möglichkeit, den aktuellen Fortschritt im Splash Screen anzuzeigen:

```
                SplashScreen.SetMessage("Schritt " + i.ToString());
            }
```

Sind alle Ladeaktivitäten absolviert, sollten wir auch unseren Splash Screen wieder ausblenden:

```
            SplashScreen.Stop();
        }
```

Beim Start der Anwendung sollte jetzt zunächst unser Splash Screen erscheinen und anschließend *Form1*.

HINWEIS: Man kann sicher auch die Anzeige des Splash Screens in einen extra Thread auslagern, um die Aktualisierungen dieser Forms unabhängig von den anderen Aktivitäten der Anwendung zu realisieren. Die Alternative ist das regelmäßige Aufrufen der *DoEvents*-Methode, wie in obigem Beispiel demonstriert.

31.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen

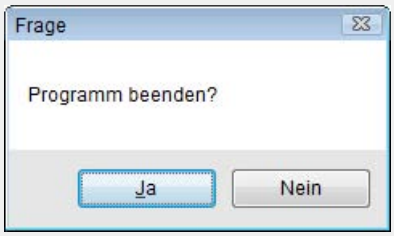
Für das Schließen der Anwendung bieten sich viele Möglichkeiten und so ist es sicher ratsam, die entsprechende Routine zentral zu organisieren. Dazu bietet sich das *FormClosing*-Ereignis an, wie es das folgende Beispiel zeigt.

BEISPIEL 31.11: Sicherheitsabfrage vor dem Schließen des Formulars

C#

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    e.Cancel = MessageBox.Show("Programm beenden?", "Frage", MessageBoxButtons.YesNo) ==
                                        DialogResult.No;
}
```

Ergebnis



Die Neugier des Programmierers ist in vielen Fällen angebracht, und so bietet der Parameter *e* ganz nebenbei auch Informationen darüber, warum das Ereignis ausgelöst wurde. Verantwortlich dafür ist der Member *CloseReason*, dessen einzelne Konstanten Sie der folgenden Tabelle entnehmen können:

Member	Beschreibung
ApplicationExitCall	Die Methode *Application.Exit* wurde im Programm aufgerufen.
FormOwnerClosing	Das Hauptformular wurde geschlossen.
MdiFormClosing	Das zentrale MDI-Form wurde geschlossen.
None	Hier weiß auch die API nicht weiter.
TaskManagerClosing	Der Taskmanager will die Anwendung schließen.
UserClosing	Ein Nutzeraktion (Formularschaltflächen) führt zum Schließen.
WindowsShutDown	Das Betriebssystem wird heruntergefahren.

31.2.4 Ein Formular durchsichtig machen

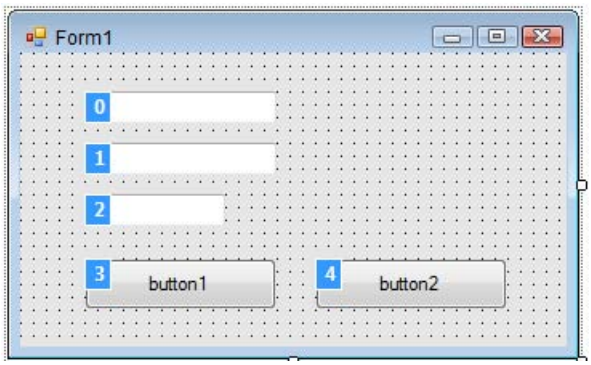
Sind Ihnen die bisherigen Formulare zu schlicht, können Sie Ihre Anwendung auch mit einem Transparenzeffekt aufpeppen. Setzen Sie dazu einfach die *Opacity*-Eigenschaft auf einen Wert zwischen 0% (vollständige Transparenz) und 100%. Das Ergebnis bei 50% zeigt folgender Bildschirmausschnitt mit Blick auf den Desktop:

31.2.5 Die Tabulatorreihenfolge festlegen

Gerade bei Dialogboxen ist die Eingabereihenfolge von übergeordnetem Interesse. Was nützt dem Anwender eine Dialogbox, in der der Eingabefokus willkürlich zwischen den Text- und Combo-Boxen hin und her springt?

Da der Wechsel von einem Eingabe-Control zum nächsten mit der Tabulatortaste erfolgt, spricht man auch von Tabulatorreihenfolge. Jedes sichtbare Steuerelement verfügt zu diesem Zweck über die Eigenschaften *TabIndex* und *TabStop*. Während mit *TabStop* lediglich festgelegt wird, ob das Control überhaupt den Fokus erhalten kann (mittels Tab-Taste), können Sie mit *TabIndex* Einfluss auf die Reihenfolge nehmen.

Visual Studio unterstützt Sie bei dieser Arbeit recht gut. Um den Überblick zu verbessern, können Sie die Tabulatorreihenfolge im Entwurfsmodus sichtbar machen. Aktivieren Sie diese über den Kontextmenüpunkt *Ansicht/Aktivierreihenfolge*. Ihr Formular dürfte danach zum Beispiel folgenden Anblick bieten:



Klicken Sie jetzt einfach in der gewünschten Reihenfolge in die kleinen Fähnchen, um die Tabulatorreihenfolge zu ändern.

31.2.6 Ausrichten von Komponenten im Formular

War es in der ersten Version von Visual Studio noch eine Qual bzw. ein riesiger Aufwand, Komponenten in einem Formular sauber auszurichten, stellt dies heutzutage kaum noch ein Problem dar. Zwei wesentliche Verfahren bieten sich an:

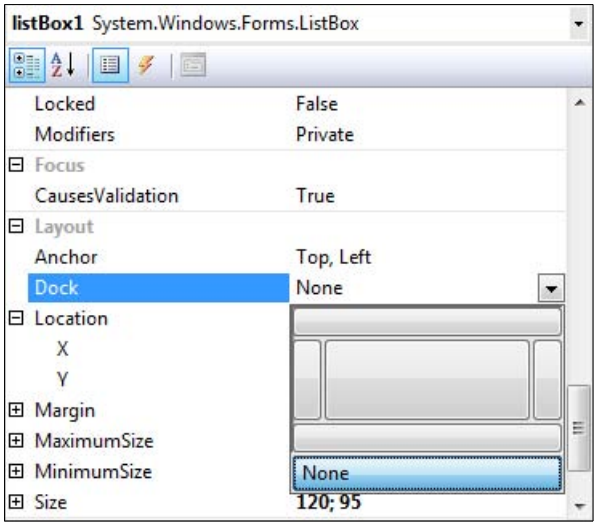
- **Docking** (Andocken an die Außenkanten einer übergeordneten Komponente)
- **Anchoring** (Ausrichten relativ zu den Außenkanten einer übergeordneten Komponente)

Verantwortlich für das Ausrichten sind die beiden nahe liegenden Eigenschaften *Dock* und *Anchor*.

HINWEIS: Natürlich wollen wir der Vollständigkeit halber nicht vergessen, hier auch auf die Eigenschaften *Location* (linke obere Ecke) und *Size* (Breite, Höhe) hinzuweisen.

Dock

Öffnen Sie das Eigenschaftenfenster (F4) und wählen Sie die Eigenschaft *Dock*, steht Ihnen der Eigenschafteneditor zur Verfügung:

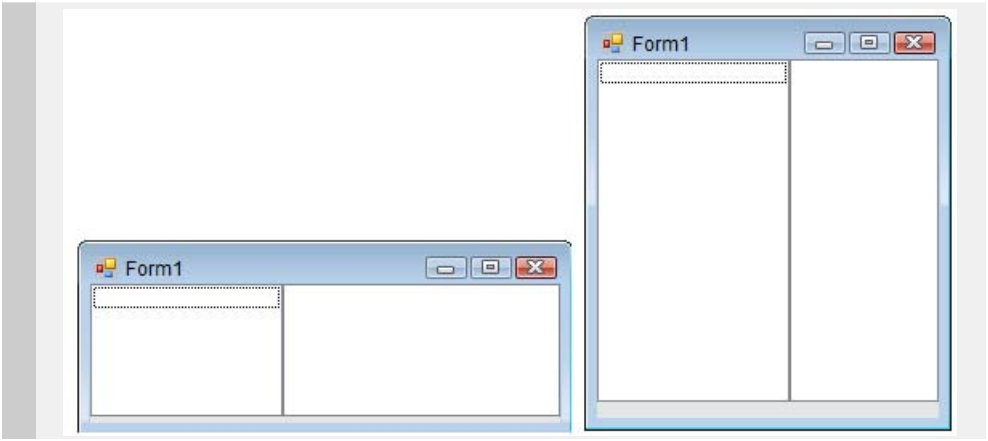


Eine Komponente lässt sich mit dieser Eigenschaft fest an den vier Außenkanten oder in der verbleibenden Clientfläche ausrichten. Dabei verändert sich die Größe der Komponente nur so, dass die Ausrichtung an den Außenkanten erhalten bleibt.

BEISPIEL 31.12: Ausrichten zweier Listboxen in einem Fenster

Die linke *ListBox* ändert lediglich ihre Höhe, die rechte *ListBox* füllt immer den gesamten freien Formbereich aus.

BEISPIEL 31.12: Ausrichten zweier Listboxen in einem Fenster



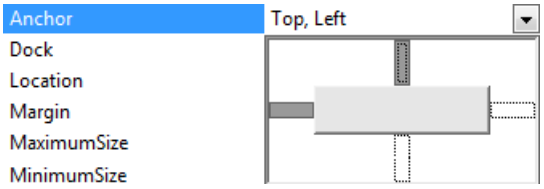
HINWEIS: Möchten Sie die Ausrichtung aufheben, wählen Sie im Eigenschaften-Editor die unterste Schaltfläche (*None*).

Damit stellt jetzt auch das Positionieren von Bildlaufleisten kein Problem mehr dar, einfach die Komponenten am rechten bzw. am unteren Rand ausrichten.

HINWEIS: Mit der Formulareigenschaft *Padding* können Sie einen Mindestabstand beim Docking vorgeben. Auf diese Weise lassen sich Ränder zu den Formularkanten definieren.

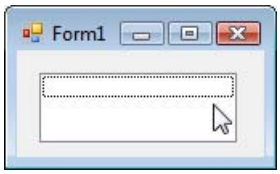
Anchor

Etwas anders als die *Dock*-Eigenschaft verhält sich die *Anchor*-Eigenschaft. Standardmäßig ist *Anchor* immer mit *Left, Top* aktiv, bei Größenänderungen des Formulars bleibt also der Abstand der Komponente zum linken und oberen Rand der umgebenden Komponente immer gleich. Auch hier steht ein eigener Eigenschafteneditor zur Verfügung:



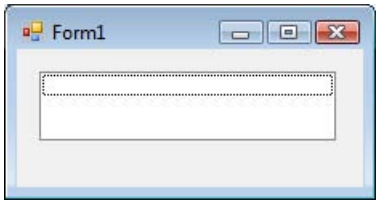
Die folgenden Abbildungen zeigen Ihnen die Auswirkungen verschiedener *Anchor*-Einstellungen auf ein Formular:

Original Fenstergröße

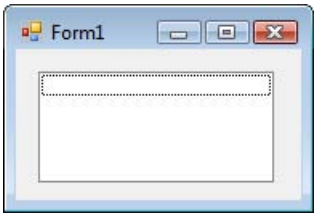


Fenster vergrößert (*Anchor=Left,Top*)

Fenster vergrößert
(*Anchor=Left,Top,Right*)



Fenster vergrößert
(*Anchor=Left,Top,Right,Bottom*)



Zur Laufzeit können Sie die *Anchor*-Eigenschaft mit den Werten der Enumeration *AnchorStyles* festlegen (siehe folgende Tabelle).

Konstante	Das Steuerelement ist ...
None	... nicht verankert.
Top	... am oberen Rand verankert.
Bottom	... am unteren Rand verankert.
Left	... am linken Rand verankert.
Right	... am rechten Rand verankert.

Da die *AnchorStyles*-Enumeration über ein *[Flags]*-Attribut verfügt, können die einzelnen Konstanten bitweise verknüpft werden.

BEISPIEL 31.13: Herstellen der Standardverankerung eines Button

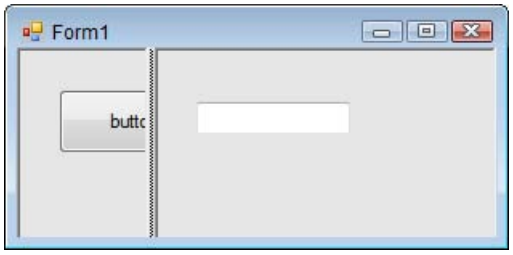
C#
```
button1.Anchor = AnchorStyles.Left | AnchorStyles.Top;
```

31.2.7 Spezielle Panels für flexible Layouts

Auch hier geht es um die Bereitstellung eines flexiblen Formularlayouts. Allerdings haben wir es, im Unterschied zum vorhergehenden Abschnitt, mit keinen Formulareigenschaften mehr zu tun, sondern mit speziellen Panels, die Sie im "Container"-Segment der Toolbox vorfinden.

SplitContainer

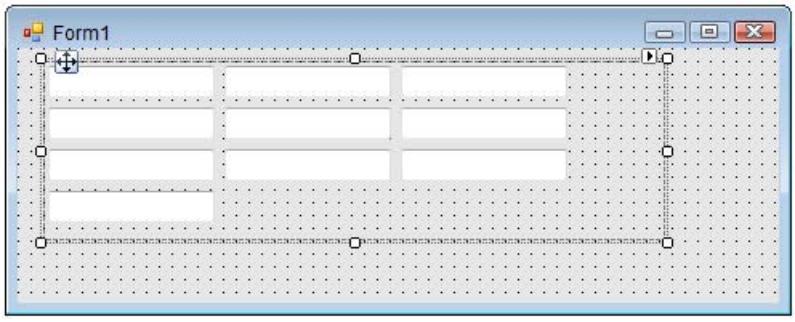
Diese Komponente gilt als Nachfolger für den *Splitter* und besteht aus zwei Panels, die durch einen zur Laufzeit veränderlichen Balken getrennt sind.



Wichtige Eigenschaften sind *Orientation* (horizontale oder vertikaler Trennbalken), *IsSplitterFixed* (fester oder beweglicher Balken) und *BorderStyle* (in der Standardeinstellung *None* bleibt der Balken unsichtbar).

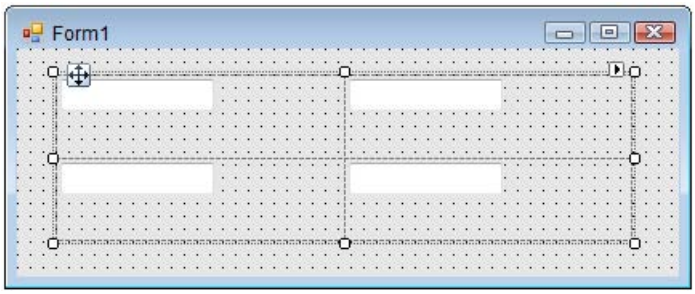
FlowLayoutPanel

Diese Komponente layoutet die auf ihr platzierten Steuerelemente dynamisch in horizontaler oder vertikaler Richtung, d.h., bei einer Größenänderung des Formulars werden die Steuerelemente automatisch "umgebrochen" (*WrapContents = True*). Wichtig ist die *Anchor*-Eigenschaft, die die Ränder des Containers (*Top*, *Bottom, Left, Right*) bestimmt, an welche die Steuerelemente gebunden werden sollen.



TableLayoutPanel

Diese Komponente besorgt ein dynamisches Layout, welches sich an einer Gitterstruktur orientiert. Wichtig sind die *Rows*- und die *Columns*-Eigenschaft, die über spezielle Dialoge (aufrufbar über den Smarttag rechts oben) den individuellen Bedürfnissen angepasst werden können.



31.2.8 Menüs erzeugen

An dieser Stelle wollen wir etwas vorgreifen, da *MenuStrip*- und *ContextMenuStrip*-Komponente (Nachfolger der veralteten *MainMenu*- und *ContextMenu*-Controls) eigentlich erst in das folgende Kapitel gehören, wo es um die Beschreibung der wichtigsten Steuerelemente geht. Doch die optische Verzahnung zwischen Menü und Formular ist so eng, dass wir bereits an dieser Stelle auf dieses Thema eingehen wollen.

Beide Komponenten haben gegenüber dem veralteten *MainMenu*/*ContextMenü* wesentlich verbesserte Features. So kann jedem Menüelement über seine *Image*-Eigenschaft auf einfache Weise eine Grafik zugewiesen werden. Menüeinträge können auch als *ComboBox*, *TextBox* oder *Separator* in Erscheinung treten.

MenuStrip

Möchten Sie ein "normales" Menü erzeugen, platzieren Sie einfach eine *MenuStrip*-Komponente auf das Formular. Es handelt sich zunächst um eine nicht sichtbare Komponente, die lediglich im Komponentenfach zu sehen ist. Doch halt, auch in der Kopfzeile des Formulars tut sich etwas, sobald die Komponente markiert wird: Das Menü wird genau dort bearbeitet, wo es sich zur Laufzeit auch befindet. Klicken Sie also in den Text "Hier eingeben" und tragen Sie beispielsweise "Datei" ein. Automatisch werden bereits zwei weitere potenzielle Menüpunkte erzeugt (ein Menüpunkt auf der gleichen Ebene, ein untergeordneter Menüpunkt), die Sie auf die gleiche Weise bearbeiten können.

HINWEIS: Möchten Sie einen Trennstrich einfügen, genügt die Eingabe eines einzelnen Minuszeichens (-) als Beschriftung!

BEISPIEL 31.14: MenuStrip

C#

Die Abbildung am Ende des Beispiels zeigt ein Datei-Menü mit zwei Einträgen (*MenuItem*s). Dazwischen befindet sich ein Trennstrich. Die kleine Grafik bei *Datei/Öffnen* wurde diesem Menüpunkt über dessen *Image*-Eigenschaft zugewiesen.

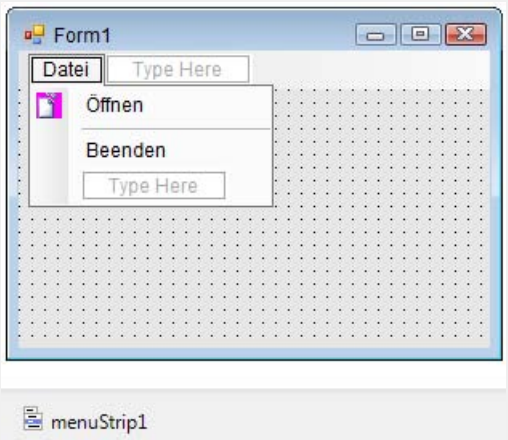
Um für einen Menüpunkt die Ereignisprozedur zu erzeugen genügt ein Doppelklick, und schon befinden Sie sich wieder im Code-Editor und programmieren die Funktionalität.

BEISPIEL 31.14: MenuStrip

Die beiden Menüeinträge werden mit Code hinterlegt.

```
public partial class Form1 : Form
{ ...
    private void öffnenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        MessageBox.Show("Das ist nur ein Test!");
    }

    private void beendenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Ergebnis



Anstatt eines normalen Menüeintrags können auch eine *TextBox* oder eine *ComboBox* verwendet werden.

BEISPIEL 31.15: *ComboBox* in *MenuStrip*

C#

Die oberste Menüebene des Vorgängerbeispiels wird um eine *ComboBox* erweitert, mit welcher man die Hintergrundfarbe des Formulars einstellen kann.

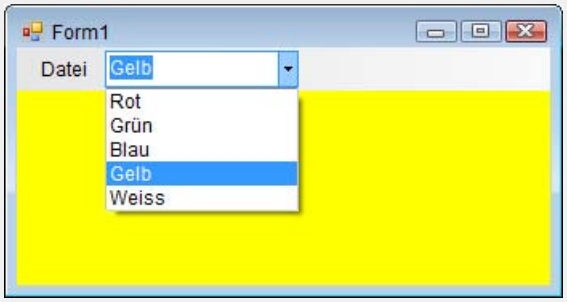
Die *Items*-Eigenschaft der *ComboBox* wird mit dem über das Eigenschaftenfenster erreichbaren Editor zeilenweise mit den Einträgen für die einzelnen Farben gefüllt. Zur Programmierung kann man das *SelectedIndexChanged*-Ereignis der *ComboBox* ausnutzen:

```
private void toolStripComboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Color c = Color.White;
    switch (toolStripComboBox1.SelectedIndex)
    {
```

BEISPIEL 31.15: *ComboBox* in *MenuStrip*

```
        case 0: c = Color.Red; break;
        case 1: c = Color.Green; break;
        case 2: c = Color.Blue; break;
        case 3: c = Color.Yellow; break;
        case 4: c = Color.White; break;
    }
    this.BackColor = c;
}
```

Ergebnis



ContextMenuStrip

Die Programmierung eines Kontextmenüs unterscheidet sich nur unwesentlich von der eines normalen Menüs. Der Entwurfsprozess ist nahezu identisch.

HINWEIS: Da sich ein Kontextmenü in der Regel auf ein bestimmtes visuelles Steuerelement bezieht, verfügen letztere über eine *ContextMenuStrip*-Eigenschaft.

BEISPIEL 31.16: *ContextMenuStrip*

C#

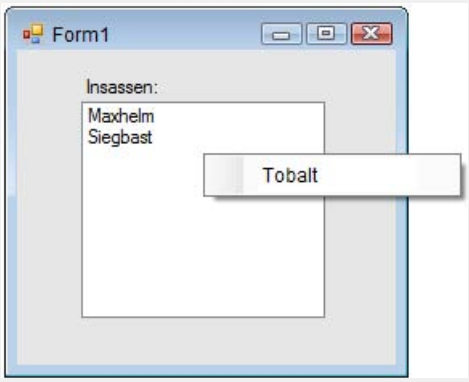
Eine *ContextMenuStrip*-Komponente wird mit einer *TextBox* ausgestattet, um zur Laufzeit eine *ListBox* mit Einträgen zu füllen. Die *ContextMenuStrip*-Eigenschaft der *ListBox* ist mit dem Kontextmenü zu verbinden!

Nach Klick mit der rechten Maustaste auf die *ListBox* erscheint das Kontextmenü. Jeder neue Eintrag ist mittels *Enter*-Taste abzuschließen:

```
private void toolStripTextBox1_KeyUp(object sender, KeyEventArgs e)
{
    if (e.KeyCode == Keys.Enter)
    {
        listBox1.Items.Add(toolStripTextBox1.Text);
        toolStripTextBox1.Text = String.Empty;
    }
}
```

BEISPIEL 31.16: *ContextMenuStrip*

Ergebnis



Weitere Eigenschaften von Menüeinträgen

Jeder Menüeintrag wird durch ein *ToolStripMenuItem*-Objekt dargestellt, hat also eigene Eigenschaften, von denen die *Text*-Eigenschaft die offensichtlichste ist. Auch viele der übrigen Eigenschaften erklären sich von selbst.

BEISPIEL 31.17: Mit der Eigenschaft *Checked* bestimmen Sie, ob vor dem Menüpunkt ein Häkchen angezeigt werden soll oder nicht.

C#

In der Ereignisprozedur können Sie den Wert entsprechend setzen oder auslesen:

```
private void testToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (testToolStripMenuItem.Checked)
    {
        MessageBox.Show("Das ist nur ein Test!");
        testToolStripMenuItem.Checked = false;
    }
}
```

Über die Eigenschaft *Visible* steuern Sie die Sichtbarkeit des Menüeintrags. Besser als das Ausblenden ist meist das Deaktivieren mit der *Enabled*-Eigenschaft. Über *ShortcutKeys* bzw. *ShowShortcutKeys* können Sie den Menüpunkt mit einer Tastenkombination verbinden (z.B. Alt+G oder F7).

31.3 MDI-Anwendungen

Ein Windows-Programm läuft oft als MDI-Applikation ab, d h., innerhalb eines Rahmen- bzw. Mutterfensters können sich mehrere so genannte Kindfenster tummeln[1]. Diese werden vom Mutterfenster verwaltet und so "an der Leine gehalten", dass sie z.B. auch dessen Bereich nicht verlassen können.

31.3.1 "Falsche" MDI-Fenster bzw. Verwenden von Parent

Formulare verfügen, wie auch die einfachen Controls, über die Eigenschaft *Parent*. Mit Hilfe dieser Eigenschaft können Sie ein Formular wie ein Control behandeln und in ein übergeordnetes Formular einfügen.

Das Resultat dieses Vorgehens: Das Formular kann den Clientbereich seines Parent nicht mehr verlassen, Sie können es aber wie gewohnt verschieben, skalieren oder auch schließen. Von besonderem Interesse dürfte die Möglichkeit sein, die untergeordneten Formulare wie Controls anzudocken. Damit steht der Programmierung von Toolbars etc. kaum noch etwas im Wege.

BEISPIEL 31.18: Einfügen des Formulars *Form2* in den Clientbereich von *Form1*

C#

```
public partial class Form1 : Form
{ ...
    private void button1_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.TopLevel = false;
        f2.Parent = this;
        f2.Show();
    }
}
```

In *Form2* realisieren wir das Docking:

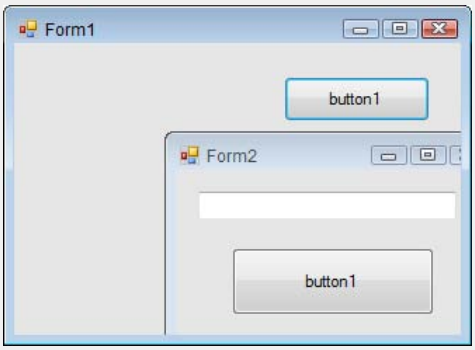
```
public partial class Form2 : Form
{ ...
    private void button1_Click(object sender, EventArgs e)
    {
        this.Dock = DockStyle.Left;
    }
}
```

[1] Zum Beispiel Textverarbeitungsprogramme sind meist als Multiple Document Interface-Applikation aufgebaut. Die einzelnen Dokumente sind die Kindfenster.

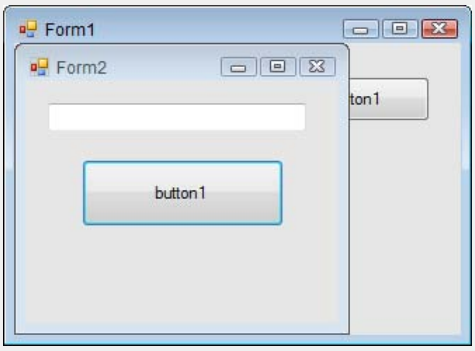
BEISPIEL 31.18: Einfügen des Formulars *Form2* in den Clientbereich von *Form1*

Ergebnis

Das Endergebnis unserer Bemühungen:

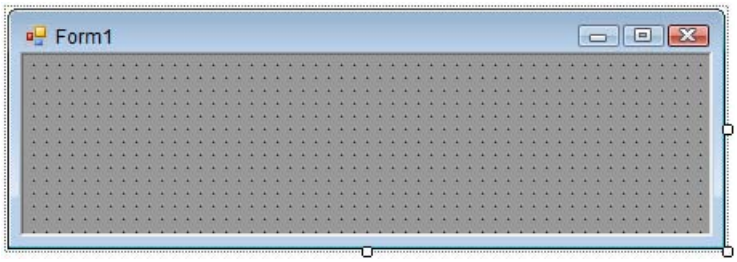


bzw. angedockt:



31.3.2 Die echten MDI-Fenster

Möchten Sie "echte" MDI-Anwendungen programmieren, brauchen Sie wie im vorhergehenden Beispiel ebenfalls mindestens zwei Formulare, von denen jedoch eines als MDIContainer definiert ist (Eigenschaft *IsMdiContainer=True*). Nachfolgend ändert sich schon zur Entwurfszeit das Aussehen des Formulars:

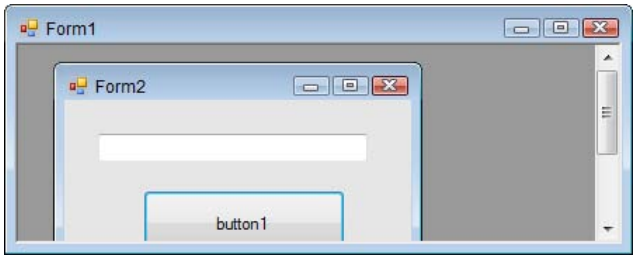


HINWEIS: Sie sollten keine weiteren Controls im MDIContainer platzieren, es sei denn, Sie richten diese mit Hilfe der *Dock*-Eigenschaft an den Außenkanten des Formulars aus (z.B. ein *Panel*).

31.3.3 Die Kindfenster

Die Kind- oder auch Child-Fenster werden über die Eigenschaft *MdiParent* kenntlich gemacht. Weisen Sie dieser Eigenschaft ein MDIContainer-Fenster zu, werden die Kindfenster automatisch in den Clientbereich des MDIContainers verschoben.

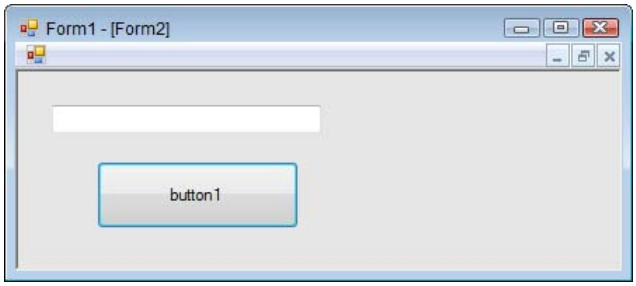
HINWEIS: Die *MDIParent*-Eigenschaft lässt sich nur zur Laufzeit zuweisen (also nicht über das Eigenschaftenfenster)!



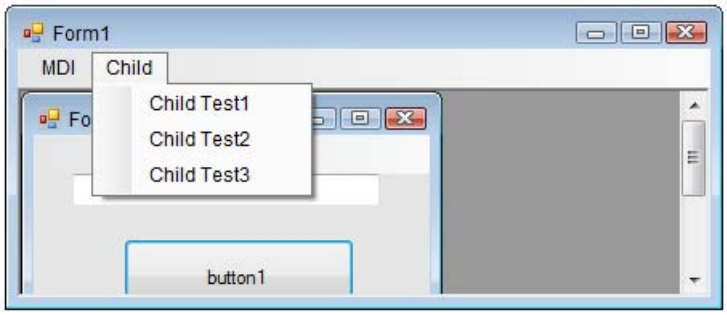
Die Umsetzung:

```
public partial class Form1 : Form
{ ...
    private void neuesChildToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Form2 f2 = new Form2();
        f2.MdiParent = this;
        f2.Show();
    }
}
```

Vergrößern Sie ein MDI-Kindfenster auf Vollbild, so erscheint dessen Titel eingefasst in eckigen Klammern neben dem Titel des Hauptfensters:



Verfügen beide Fenster über eigene Menüs, so werden diese standardmäßig im Menü des MDI-Containers kombiniert:

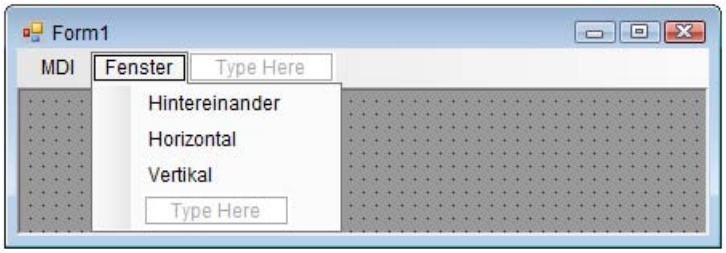


Gegenüber ihren konventionellen Kollegen besitzen die MDI-Kindfenster einige Einschränkungen, die durch das MDI-Konzept bedingt sind:

- MDI-Kindfenster werden nicht in der Windows-Taskleiste angezeigt.
- MDI-Kindfenster können den Clientbereich des MDIContainers nicht verlassen. Verschieben Sie die Fenster aus dem sichtbaren Bereich des MDIContainers, werden automatisch die nötigen Bildlaufleisten im Container angezeigt.
- Maximieren Sie ein MDI-Kindfenster, erreicht dieses maximal die Größe des verbleibenden Clientbereichs des MDIContainers (abzüglich Statuszeile, Menüleiste und Toolbar). Die Schaltflächen für Maximieren, Minimieren und Schließen des Kindfensters werden in diesem Fall in die Menüleiste des MDIContainers eingefügt.

31.3.4 Automatisches Anordnen der Kindfenster

Ein MDIContainer verfügt über die Methode *LayoutMdi,* mit der die vorhandenen Kindfenster nebeneinander, überlappend oder als Symbole angeordnet werden können. Üblicherweise entspricht dies dem in vielen MDI-Anwendungen vorhandenen *Fenster*-Menü:



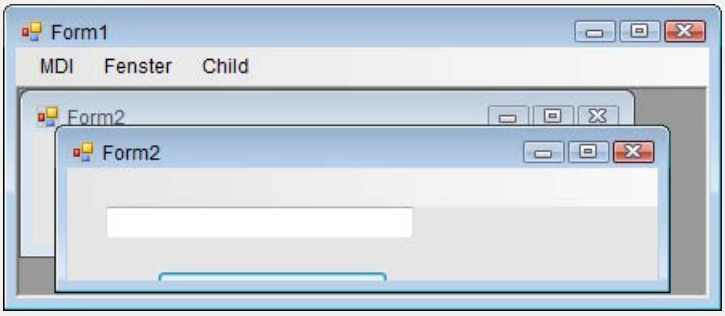
BEISPIEL 31.19: Hintereinander anordnen

C#
```
private void hintereinanderToolStripMenuItem_Click(object sender, EventArgs e)
{
```

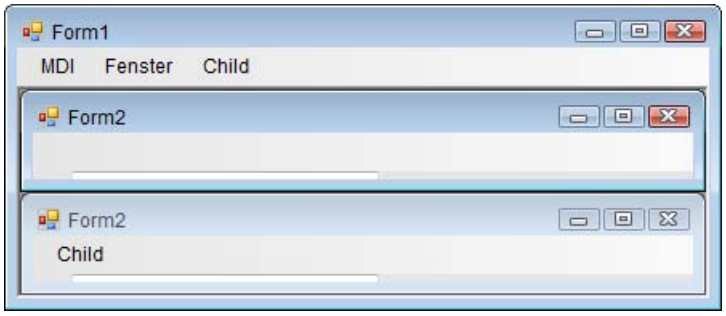
BEISPIEL 31.19: Hintereinander anordnen

```
    this.LayoutMdi(MdiLayout.Cascade);
}
```

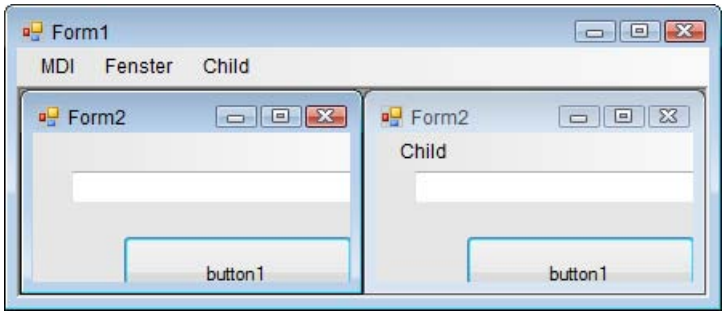
Ergebnis



Horizontal anordnen:



Vertikal anordnen:



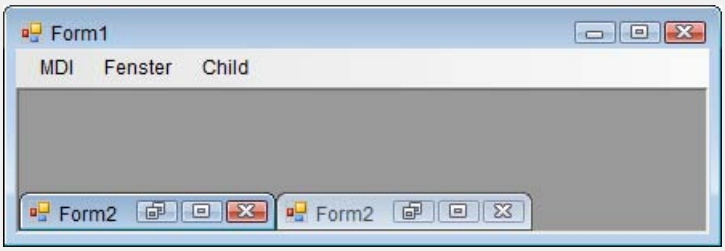
31.3.5 Zugriff auf die geöffneten MDI-Kindfenster

Über die Collection *MdiChildren* bietet sich die Möglichkeit, alle vorhandenen Kindfenster des MDIContainers zu verwalten.

BEISPIEL 31.20: Minimieren aller MDI-Kindfenster

C#
```
private void minimierenToolStripMenuItem_Click(object sender, EventArgs e)
{
    foreach (Form f in this.MdiChildren)
    {
        f.WindowState = FormWindowState.Minimized;
    }
}
```

Ergebnis



31.3.6 Zugriff auf das aktive MDI-Kindfenster

Neben der Liste aller Kindfenster ist meist auch das gerade aktuelle von besonderem Interesse für den Programmierer, muss er doch häufig auf Inhalte des aktuellen Fensters (z.B. Textfelder, Markierungen) zugreifen. Die per MDIContainer verfügbare Eigenschaft *ActiveMdiChild* lässt die aufkommende Freude recht schnell vergessen, liefert diese doch zunächst nur ein Objekt der Klasse *Form* zurück. Es ist also Ihre Aufgabe, das zurückgegebene Objekt entsprechend zu typisieren.

BEISPIEL 31.21: Abfrage und Typisieren des aktuellen MDI-Kindfensters per MDIContainer

C#
```
if (this.ActiveMdiChild != null)
{
    myMDIForm mdifrm = (myMDIForm)(this.ActiveMdiChild);
    mdifrm.label1.Text = "Ich bin aktiv";
}
```

31.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü

Das Vermischen der Menüs von MDIContainer und MDI-Kindfenstern kann über spezielle Eigenschaften *(AllowMerge*, *MergeAction*, *MergeIndex)* gesteuert werden und ist (fast) eine "Wissenschaft" für sich.

AllowMerge

Die Einstellungen der Eigenschaften *MergeAction* und *MergeIndex* wirken sich nur dann aus, wenn für den *MenuStrip* der Wert von *AllowMerge* auf *true* gesetzt ist, was standardmäßig der Fall ist. Setzen Sie den Wert auf *false*, erfolgt keine Mischung der Menüs.

MergeAction und MergeIndex

Mit der Eigenschaft *MergeAction* geben Sie vor, wie die Menüleisten von Container und Kind kombiniert werden, *MergeIndex* spezifiziert in einigen Fällen die Position des einzufügenden Menüelements.

HINWEIS: Ein umfangreiches Beispiel dazu (Kleines Textverarbeitungsprogramm) finden Sie in unserem [Visual C# Kochbuch].

31.4 Praxisbeispiele

31.4.1 Informationsaustausch zwischen Formularen

In diesem Beispiel wird das auch von fortgeschrittenen Programmierern häufig nachgefragte Thema "Wie greife ich von FormA auf FormB zu" abgehandelt.

Überblick

Unter Visual Studio wird der Code zum Erzeugen des Haupt- bzw. Startformulars (standardmäßig *Form1*) von der IDE automatisch generiert. Um das Erzeugen weiterer Formulare nach dem Muster

```
Form2 f2 = new Form2();
```

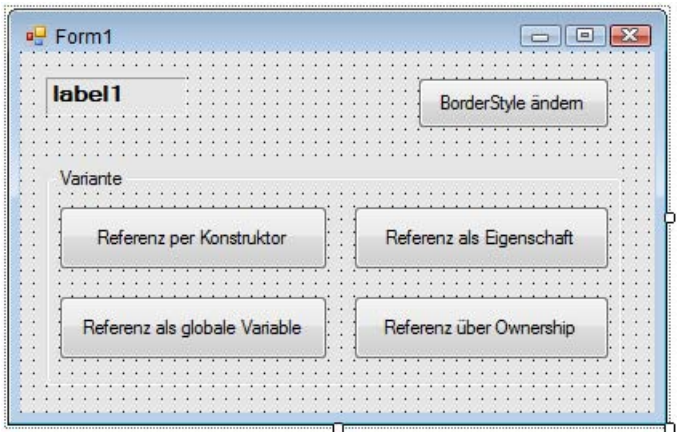
... muss sich der Programmierer selbst kümmern. Existiert die Formularinstanz, so können Sie gemäß den Regeln der OOP auf deren öffentliche Mitglieder auch von außerhalb zugreifen. Wenn beispielsweise im Code von *Form1* ein weiteres Formular (*Form2*) instanziiert wird, so ist es kein Problem, die öffentliche Member von *Form2* zu verwenden. Etwas komplizierter wird es in umgekehrter Richtung, also wenn Sie von *Form2* (oder weiteren Kindformularen *Form3*, *Form4*, ...) auf das Hauptformular (oder andere Kindformulare) zugreifen wollen. Wir wollen folgende Varianten betrachten, die eine Beziehung zwischen *Form2* (Childform) und *Form1* (Parentform) ermöglichen:

- Erzeugen einer Referenz auf *Form1* im Konstruktor von *Form2*.
- Setzen einer Eigenschaft in *Form2*, die eine Referenz auf *Form1* einrichtet.
- Erzeugen einer globalen Variablen die *Form1* referenziert und die von *Form2* (und weiteren untergeordneten Forms) benutzt werden kann.
- verwenden des Owner-Mechanismus der Formulare.

In unserem Beispiel, für welches wir fünf Formulare benötigen, werden wir diese vier Varianten vergleichen. Das erste Formular (*Form1*) wird als Hauptformular dienen und die anderen vier (*Form2*, *Form3*, *Form4*, *Form5*) sollen die untergeordneten Formulare sein. Letztere sind – der besseren Vergleichsmöglichkeiten wegen – mit der gleichen Bedienoberfläche ausgestattet und arbeiten auf identische Weise mit dem Hauptformular zusammen.

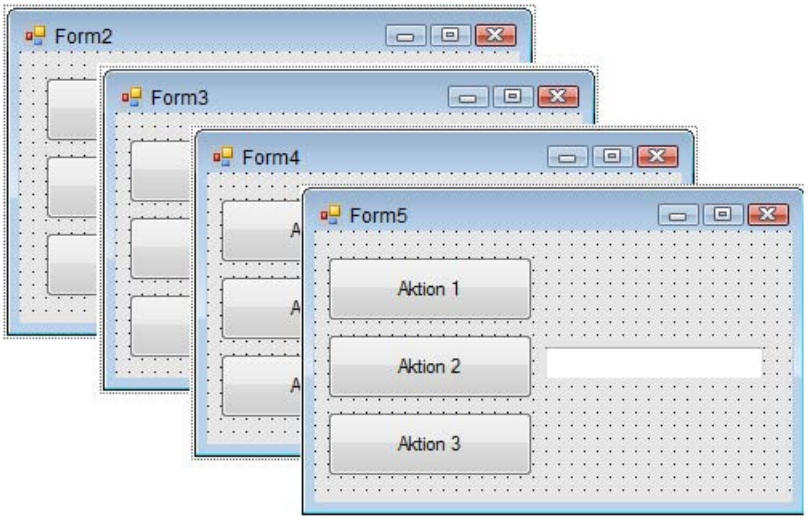
Bedienoberfläche Form1 (Hauptformular)

Öffnen Sie ein neues C#-Projekt als Windows Forms-Anwendung und gestalten Sie die abgebildete Oberfläche. Die in der *GroupBox* angeordneten Schaltflächen (*button2*, ..., *button5*) rufen je ein untergeordnetes Formular (*Form2*, ..., *Form5*) auf, welches die gewünschte Variante demonstriert, wobei auf die beiden oben angeordneten Controls (*label1*, *button1*) des Hauptformulars zugegriffen wird.



Bedienoberfläche Form2 ... Form5 (untergeordnete Formulare)

Über das Menü *Projekt|Windows-Form hinzufügen...* ergänzen Sie das Projekt um vier weitere Formulare mit identischer Oberfläche (drei *Button*s und eine *TextBox*):



Die drei *Button*s sollen diverse Aktionen auf dem Hauptformular *Form1* auslösen:

- *button1* führt eine Instanzenmethode aus (Anzeige von Datum/Uhrzeit),
- *button2* soll *label1* mit dem Inhalt von *textBox1* füllen und
- *button3* löst das *Click*-Event für *button1* aus, wodurch der *BorderStyle* von *Form1* geändert wird.

Allgemeiner Code für Form1

Zunächst treffen wir im Code von *Form1* einige Vorbereitungen für den späteren Zugriff:

```
public partial class Form1 : Form
{
        public Form1()
        {
            InitializeComponent();
        }
```

Den Lesezugriff auf die von außerhalb zu manipulierenden Controls als Eigenschaften offenlegen:

```
        internal Label lbl1
        {
            get
            { return label1;  }
        }

        internal Button btn1
        {
            get
            { return button1; }
        }
```

Irgendeine Methode:

```
        internal void machWas()
        {
            this.Text = DateTime.Now.ToString();
        }
```

Irgendeine Click-Aktion:

```
        private void button5_Click_1(object sender, EventArgs e)
        {
            if (this.FormBorderStyle.Equals(FormBorderStyle.None))
                this.FormBorderStyle = FormBorderStyle.Sizable;
            else
                this.FormBorderStyle = FormBorderStyle.None;
        }
        ...
}
```

Den Ereigniscode für *button1 ... button4*, in welchem die untergeordneten Formulare aufgerufen werden, ergänzen wir später.

Variante 1: Übergabe der Formular-Referenz im Konstruktor

Bei dieser Variante wird der Konstruktor von *Form2* so modifiziert, dass er eine Referenz auf *Form1* entgegennehmen und damit die private Zustandsvariable *m_Form* setzen kann.

```
public partial class Form2 : Form
{
        private Form1 m_Form;

        public Form2(Form1 frm)
        {
            InitializeComponent();
            m_Form = frm;           // Referenz auf Hauptformular setzen
        }
```

Eine Instanzenmethode in *Form1* ausführen:

```
        private void button1_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.machWas();
            else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
```

Ein Steuerelement in *Form1* setzen:

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.lbl1.Text = textBox1.Text;
            else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
```

Ein Ereignis in *Form1* auslösen:

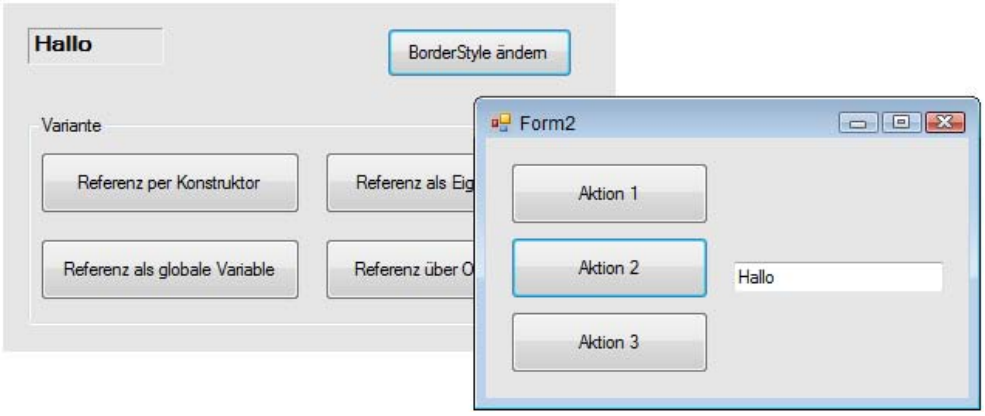
```
        private void button3_Click(object sender, EventArgs e)
        {
            if (m_Form != null)
                m_Form.btn1.PerformClick();    // Click-Event des Buttons auslösen
            else
                throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
        }
    }
```

Sie haben jetzt sicherlich bemerkt, dass alle *Click*-Events die Referenz für *m_Form* auf *null* testen und andernfalls einen Fehler auswerfen. Dies ist wichtig, weil der Benutzer von *Form2* sichergehen muss, dass eine gültige Referenz auf *Form1* vorliegt.

Nun müssen wir nur noch den Code von *Form1* ergänzen:

```
private void button1_Click(object sender, EventArgs e)
{
        Form2 frm = new Form2(this);     // Form1-Referenz übergeben!
        this.AddOwnedForm(frm);          // frm zum Besitz von Form1 hinzufügen
        frm.Show();
}
```

Starten Sie die Anwendung, so wird zunächst nur das Hauptformular angezeigt. Ein Klick auf *button1* lässt *Form2* erscheinen. Klicken Sie hier auf "Aktion 1", so werden in der Titelleiste des Hauptformulars Datum und Uhrzeit angezeigt. Geben Sie irgendetwas in die *TextBox* ein, klicken Sie "Aktion 2" und das *Label* des Hauptformulars wird den Inhalt übernehmen. Ein Klick auf "Aktion 3" lässt die Umrandung des Hauptformulars verschwinden und beim nächsten Klick wieder erscheinen.



An dieser Stelle scheint eine Bemerkung zur *AddOwnedForm()*-Methode angebracht: Sie können deren Aufruf auch weglassen, müssen dann aber zum Beispiel in Kauf nehmen, dass nach dem Minimieren des Hauptformulars das untergeordnete Formular an seinem Platz verbleibt und nicht – wie in unserem Fall – komplett verschwindet um nach Vergrößern des Hauptformulars wieder aufzutauchen. Auch liegt *Form2* immer auf *Form1*, kann also von diesem nicht verdeckt werden.

Variante 2: Übergabe der Formular-Referenz als Eigenschaft

Die zweite Variante weist sehr starke Ähnlichkeiten zur ersten Variante auf, weshalb wir uns auf eine verkürzte Darstellung beschränken können. *Form3* hält die *Form1*-Referenz ebenfalls in der privaten Zustandsvariablen *m_Form*, lediglich an die Stelle der Übergabe im Konstruktor tritt jetzt die Übergabe in einer WriteOnly-Eigenschaft *Form* vom Typ *Form1*:

```
public partial class Form3 : Form
{
        private Form1 m_Form;
        ...
        internal Form1 Form
```

```
        {
            set
            {  m_Form = value; }
        }
        ...
}
```

Der restliche Code entspricht dem von *Form2*.

Nun müssen wir nur noch das *Click*-Event für *button2* des Hauptformulars hinzufügen, damit *Form3* angezeigt und getestet werden kann. Im Vergleich zur Vorgängervariante ist eine zusätzliche Codezeile erforderlich, in welcher die *Form1*-Referenz als *Form*-Eigenschaft dem untergeordneten Formular zugewiesen wird:

```
        private void button2_Click(object sender, EventArgs e)
        {
            Form3 frm = new Form3();
            this.AddOwnedForm(frm);
            frm.Form = this;              // Eigenschaft setzen!
            frm.Show();
        }
```

Beim Testen gibt es erwartungsgemäß keinerlei Unterschiede zur ersten Variante.

Variante 3: Übergabe der Formular-Referenz als globale Variable

Fügen Sie über *Projekt|Klasse hinzufügen...* eine neue statische Klasse (*GlobalRef*) mit folgendem Code hinzu:

```
internal static class GlobalRef
{
        public static Form1 g_Form;
}
```

Die globale Variable (*g_Form*) hält die Referenz auf das Hauptformular *Form1*. Der Zugriff von *Form4* aus:

```
public partial class Form4 : Form
{
   ...
   private void button1_Click(object sender, EventArgs e)
   {
       if (GlobalRef.g_Form != null)
           GlobalRef.g_Form.machWas();
        else
           throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
   }

    private void button2_Click(object sender, EventArgs e)
    {
        if (GlobalRef.g_Form != null)
```

```
            GlobalRef.g_Form.lbl1.Text = textBox1.Text;
        else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if (GlobalRef.g_Form != null)
            GlobalRef.g_Form.btn1.PerformClick();
        else
            throw new InvalidOperationException("Form1 Instanz nicht gesetzt!");
    }
}
```

Auch hier gibt es große Ähnlichkeiten zu den beiden Vorgängerversionen. Ein nicht zu unterschätzender Vorteil ist allerdings der globale Standort der *Form1*-Referenz, die nun nicht nur von *Form4*, sondern von jedem beliebigen Formular werden kann.

Schließlich noch der Aufruf von *Form4* im Code des Hauptformulars *Form1*:

```
    private void button3_Click(object sender, EventArgs e)
    {
        Form4 frm = new Form4();
        this.AddOwnedForm(frm);
        GlobalRef.g_Form = this;      // Setzen der globalen Referenz!
        frm.Show();
    }
```

Auch der dritten Variante werden Sie keinerlei Unterschiede zu den Vorgängern feststellen.

Variante 4: Übergabe der Formular-Referenz als Ownership

Diese vierte und letzte Version ist leider weitaus weniger bekannt als die bis jetzt besprochenen Versionen. Benutzt wird das in Windows Forms (*System.Windows.-Forms.Control*-Klasse) eingebaute Ownership-Verhalten, nach welchem ein Control eines oder mehrere andere "besitzen" kann. In unserem Fall müssen wir *Form4* zum "Besitz" (Ownership) der übergeordneten *Form1* hinzufügen (auf die Merkmale von Ownership-Beziehungen wurde bereits im Zusammenhang mit der Version 1 bzw. der *AddOwnedForm()*-Methode eingegangen).

Der folgende Code für *Form5* funktioniert nur, wenn eine *Owner*-Eigenschaft vorliegt und in den korrekten Typ (*Form1*) gecastet werden kann. Anderenfalls wird ein Fehler erzeugt.

```
public partial class Form5 : Form
{
    ...
    private void button1_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
            (this.Owner as Form1).machWas();
```

```
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder ungültig!");
    }

    private void button2_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
            (this.Owner as Form1).lbl1.Text = textBox1.Text;
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder ungültig!");
    }

    private void button3_Click(object sender, EventArgs e)
    {
        if ((this.Owner != null) && (this.Owner.GetType().Name == "Form1"))
             (this.Owner as Form1).btn1.PerformClick();
        else
            throw new InvalidOperationException("Form1 Instanze nicht gesetzt oder ungültig!");
    }
}
```

Nun zum *Form1*-Code. Instanziierung und Aufruf von *Form5* sind ähnlich einfach wie bei den drei Vorgängerversionen:

```
    private void button4_Click(object sender, EventArgs e)
    {
         Form5 frm = new Form5();
         this.AddOwnedForm(frm);   // darf nicht weggelassen werden!
         frm.Show();
    }
```

Auch der Test dieser Version führt zu exakt gleichen Ergebnissen wie bei den Vorgängern.

Bemerkungen

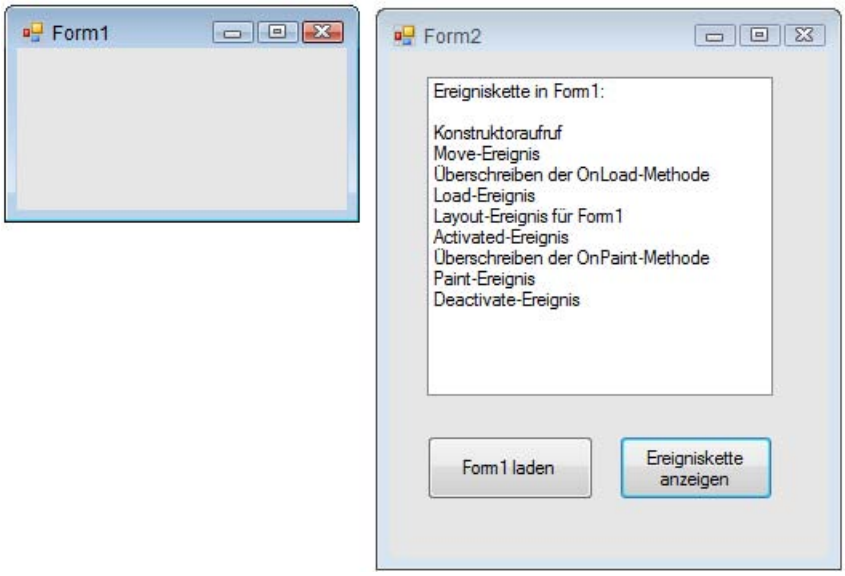
- Der Aufruf der *AddOwnedForm()*-Methode zählt bei den ersten drei Varianten zur Kür, bei der vierten Variante ist er allerdings Pflicht!
- Der ereignisgesteuerte Informationsaustausch zwischen verschiedenen Formularen bzw. Objekten auf Basis des Microsoft Event Pattern wird im Kapitel 26 ausführlich abgehandelt.

31.4.2 Ereigniskette beim Laden/Entladen eines Formulars

In welcher Reihenfolge werden die Ereignisse beim Laden bzw. Entladen eines Formulars ausgelöst? Das durch ein eigene Experiment zu erkunden wirkt nachhaltiger als das pure Auswendiglernen. Ganz nebenbei werden im vorliegenden Beispiel auch solche Fragen beantwortet: Wie ändere ich das Startformular? Wie rufe ich von *Form2* aus *Form1* auf? Wie überschreibe ich die *OnLoad*-Ereignisprozedur des Formulars?

Bedienoberfläche (Form1 und Form2)

Das das Startformular (*Form1*) hier als "Messobjekt" dient, brauchen wir noch ein zweites Formular (*Form2*), welches unser "Messgerät" aufnimmt, im konkreten Fall eine *ListBox* als Ereignislogger und zwei Buttons, mit denen das Laden von *Form1* und die Anzeige der Ereigniskette gestartet werden. *Form1* hingegen bleibt "nackt", d h., seine Oberfläche enthält hingegen keinerlei Steuerelemente (siehe Laufzeitansicht).



Warum genügt uns nicht ein einziges Formular? Die Antwort ist einfach: Durch die auf dem Formular befindlichen Steuerelemente und die in die *ListBox* vorzunehmenden Einträge vergrößert sich die Anzahl der ausgelösten Ereignisse (*Layout*, *Paint*, ...), die Sache wird unübersichtlich und eine klare Reihenfolge ist nur noch schwer zu erkennen.

Änderung des Startformulars

Nachdem Sie über das Menü *Projekt/Windows Form hinzufügen...* ein zweites Formular erzeugt haben, wollen Sie, dass dieses nach Start der Anwendung erscheint. Am einfachsten realisieren Sie das, indem Sie in der *Main*-Methode der Anwendung nicht mehr den Konstruktor von *Form1* aufrufen (normalerweise hat das Visual Studio bereits für Sie erledigt), sondern den von *Form2*. Um den Code der *Main*-Methode editieren zu können, müssen Sie im Projektmappen-Explorer auf die Datei *Program.cs* doppelklicken und dann die im Folgenden fettgedruckte Änderung vornehmen:

```
namespace WindowsFormsApplication1
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
```

```
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form2());
        }
    }
}
```

Quellcode von Form1

```
public partial class Form1 : Form
{
```

Eine generische Liste übernimmt das Zwischenspeichern des Ereignislogs. Die Liste ist *public*, damit von *Form2* aus der Inhalt auf möglichst einfache Weise gelesen werden kann:

```
    public List<string> GList;
```

Das Erzeugen der Liste erfolgt im Konstruktor des Formulars (vor Aufruf von *InitializeComponent()*!).

```
    public Form1()
    {
        GList = new List<string>();
        GList.Add("Konstruktoraufruf");

        InitializeComponent();
    }
```

Was jetzt kommt, können Sie sich leicht selbst zusammenreimen: Für jedes der interessierenden Formularereignisse schreiben Sie einen Eventhandler, dessen Rahmencode Sie wie üblich über die Ereignisse-Seite des Eigenschaftenfensters (F4) automatisch generieren lassen. Die Reihenfolge der folgenden Eventhandler ist unwichtig.

```
        private void Form1_Move(object sender, EventArgs e)
        {
            GList.Add("Move-Ereignis");
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            GList.Add("Load-Ereignis");
        }

        private void Form1_Layout(object sender, LayoutEventArgs e)
        {
           Control c = e.AffectedControl;
```

```
            GList.Add ("Layout-Ereignis für " + c.Name );
        }

        private void Form1_Activated(object sender, EventArgs e)
        {
            GList.Add("Activated-Ereignis");
        }

        private void Form1_Paint(object sender, PaintEventArgs e)
        {
            GList.Add( "Paint-Ereignis");
        }

        private void Form1_Resize(object sender, EventArgs e)
        {
            GList.Add("Resize-Ereignis");
        }

        private void Form1_FormClosing(object sender, FormClosingEventArgs e)
        {
            GList.Add("FormClosing-Ereignis");
        }

        private void Form1_FormClosed(object sender, FormClosedEventArgs e)
        {
            GList.Add("FormClosed-Ereignis");
        }

        private void Form1_Deactivate(object sender, EventArgs e)
        {
            GList.Add("Deactivate-Ereignis");
        }
```

Um herauszufinden was passiert, wenn wir die *On...*-Ereignismethoden der Basisklasse des Formulars überschreiben (siehe Bemerkungen am Schluss), fügen wir noch folgende zwei Methoden hinzu:

```
        protected override void OnLoad(EventArgs e)
        {
            GList.Add("Überschreiben der OnLoad-Methode");
            base.OnLoad(e);    // Aufruf der Basisklassenmethode
        }

        protected override void OnPaint(PaintEventArgs e)
        {
            GList.Add("Überschreiben der OnPaint-Methode");
            base.OnPaint(e);   // Aufruf der Basisklassenmethode
        }
```

Wenn Sie in obigen beiden Methoden den Aufruf der Basisklassenmethode weglassen, wird das entsprechende Ereignis nicht ausgelöst!

```
    }
}
```

Quellcode von Form2

```
public partial class Form2 : Form
{ ...
```

Zunächst brauchen wir eine Referenz auf *Form1*:

```
    Form1 frm1 = null;
```

Nach Klick auf die erste Schaltfläche wird *Form1* erzeugt und geladen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        frm1 = new Form1();
        frm1.Show();
    }
```

Ein Klick auf die zweite Schaltfläche bringt den Inhalt des Pufferspeichers *GList* aus *Form1* in der *ListBox* zur Anzeige:

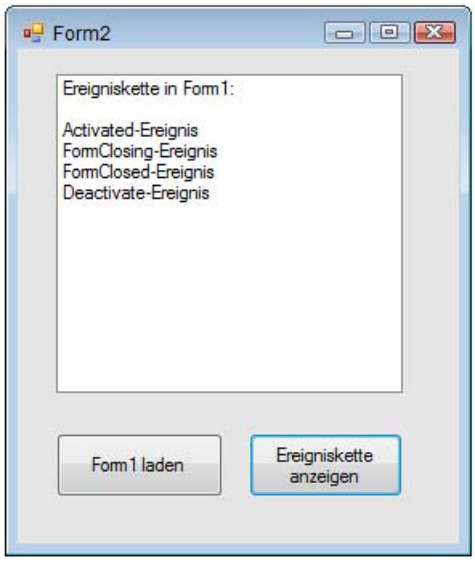
```
    private void button2_Click(object sender, EventArgs e)
    {
        listBox1.Items.Clear();
        listBox1.Items.Add("Ereigniskette in Form1:");
        listBox1.Items.Add("");
        for (int i = 0; i < frm1.GList.Count; i++)
                    listBox1.Items.Add(frm1.GList[i]);
        frm1.GList.Clear();
    }
}
```

Test

Vorhang auf für die verschiedensten Experimente! Untersuchen Sie zunächst die Ereigniskette beim Laden des Formulars (siehe obige Abbildung). Dazu klicken Sie zunächst auf den linken und dann auf den rechten Button.

Da ein Ereignis immer erst innerhalb der entsprechenden *On...*-Ereignismethode der Basisklasse ausgelöst wird, kommt in unserem Fall der in den beiden überschriebenen *On...*-Ereignismethoden enthaltene Code immer **vor** dem Code der entsprechenden Eventhandler zur Ausführung.

Nach dem Schließen von *Form1* klicken Sie erneut auf den rechten Button, um sich die Ereigniskette beim Entladen anzuschauen (siehe folgende Abbildung).



Verschieben Sie *Form1*, verändern Sie die Abmessungen, wechseln Sie den Fokus mit anderen Formularen – all diese Manipulationen finden ihren Niederschlag in einer typischen Ereigniskette.

Bemerkungen

- Beim Überschreiben von Basisklassenmethoden dürfen Sie den Aufruf der *base.On...*-Methode nicht vergessen, da ansonsten das entsprechende Ereignis nicht ausgelöst wird!
- Für das Implementieren von Initialisierungscode wird häufig der *Load*-Eventhandler des Formulars benutzt, auch die Autoren haben in den Beispielen dieses Buchs oft von dieser einfachen Möglichkeit Gebrauch gemacht. Laut Microsoft sollte man aber besser die das Ereignis auslösende *OnLoad*-Methode der Basisklasse *Form* überschreiben (siehe obiger Quellcode). Der Grund liegt im Multicast-Ereignismodell von .NET, d h., ein von einem Objekt (Subjekt) ausgelöstes Ereignis kann durchaus auch von mehreren anderen Objekten (Observern) abonniert werden, wobei die Reihenfolge der Abarbeitung unklar bzw. unübersichtlich werden kann. Ähnliche Überlegungen gelten z.B. auch für das *Paint*-Ereignis des Formulars und die das Ereignis auslösende *OnPaint*-Methode der Basisklasse (siehe dazu Grafik-Kapitel 33).

HINWEIS: Fundierte Ausführungen zum .NET-Ereignismodell finden Sie im Kapitel 26 (Microsoft Event Pattern).

Kapitel 32

Windows Forms-Komponenten

Nachdem Sie in den beiden vorhergehenden Kapiteln bereits die Grundlagen der Windows Formulare kennen gelernt haben, soll es Ziel dieses Kapitels sein, Ihnen einen Überblick über die wichtigsten visuellen Steuerelemente von Windows Forms-Anwendungen zu verschaffen. Da dieses Kapitel keine vollständige Referenz bereitstellt – diese Rolle kann die Ihnen zur Verfügung stehende Dokumentation viel effektiver übernehmen – werden nur die aus der Sicht des Praktikers wichtigsten Eigenschaften, Ereignisse und Methoden in Gestalt von Übersichten und knappen Beispielen vorgestellt.

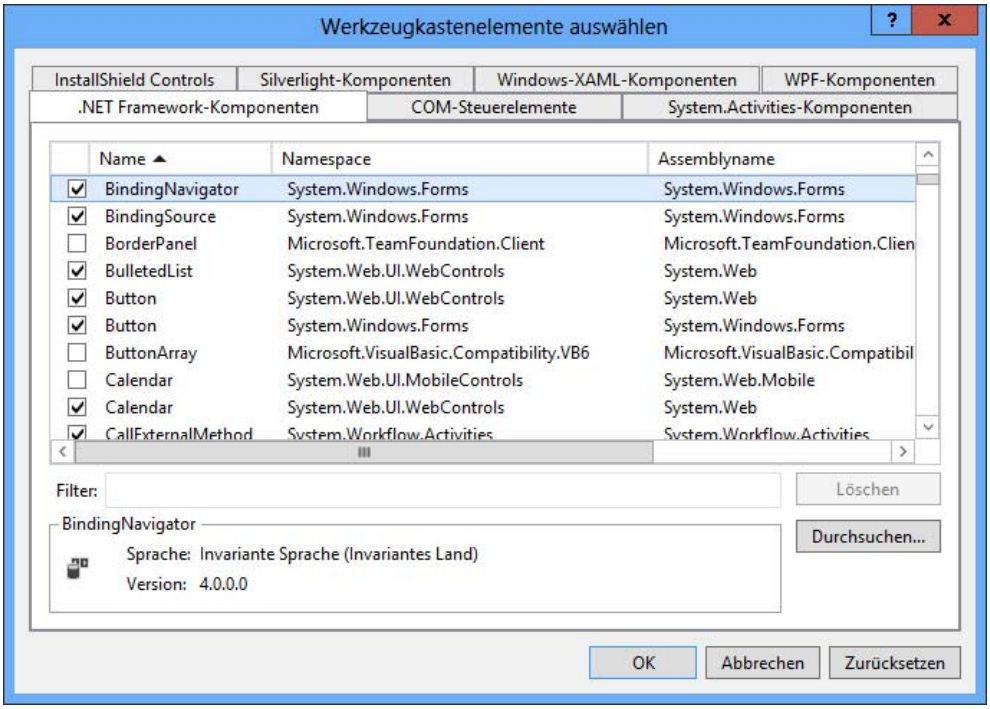
32.1 Allgemeine Hinweise

Visual Studio bietet Ihnen eine hervorragende Unterstützung bei der Entwicklung grafischer Benutzeroberflächen. Das grundlegende Handwerkszeug (Toolbox, Eigenschaftenfenster, ...) wurde Ihnen bereits im Einführungskapitel 1 im ausreichenden Umfang vermittelt. Die Vorgehensweise beim Oberflächenentwurf ist so intuitiv, dass sich weitere Erklärungen erübrigen. Stattdessen folgen einige grundsätzliche Hinweise.

32.1.1 Hinzufügen von Komponenten

Die Anzahl der Komponenten hat mit jeder neuen Version des .NET Frameworks zugenommen, sodass es aus Gründen der Übersichtlichkeit nicht mehr zu empfehlen ist, alle zusammen im Werkzeugkasten anzubieten. Außerdem wurden im Laufe der Zeit einige Steuerelemente durch neuere ersetzt (z.B. *MainMenu/ContextMenu* durch *MenuStrip/ContextMenuStrip*, *DataGrid* durch *DataGridView*, ...), die älteren Versionen müssen aber aus Gründen der Abwärtskompatibilität auch weiterhin zur Verfügung stehen, sollten aber zunächst nicht mehr in der Toolbox auftauchen.

Wenn Sie das standardmäßige Angebot ändern wollen, können Sie dem Werkzeugkasten weitere Steuerelemente hinzufügen oder welche davon entfernen. Über der entsprechenden Kategorie des Werkzeukastens klicken Sie das Kontextmenü *Elemente auswählen* ... Im Dialog "Werkzeugkastenelemente auswählen" markieren Sie die gewünschten Steuerelemente bzw. entfernen bestimmte Häkchen.



32.1.2 Komponenten zur Laufzeit per Code erzeugen

Grundsätzlich haben Sie jederzeit die Möglichkeit, auf die Dienste des visuellen Designers zu verzichten und stattdessen den Code zur Erzeugung des Steuerelements eigenhändig zu schreiben. Obwohl das scheinbar mit einiger Mehrarbeit verbunden ist, kann es in einigen Fällen durchaus sinnvoll sein (z.B. beim Erzeugen von Steuerelemente-Arrays).

BEISPIEL 32.1: Ein Array mit drei Schaltflächen per Code erzeugen

C#

```
public partial class Form1 : Form
{
```

Die globalen Deklarationen:

```
    private const int n = 3;                  // Anzahl der Buttons
    private const int xpos = 10, ypos = 10;   // linke obere Ecke des ersten Buttons

    private Button[] buttons = new Button[n]; // Array, welches Platz für n Buttons bietet
```

Steuerelemente werden im Konstruktorcode erzeugt:

```
    public Form1()
    {
        InitializeComponent();
```

BEISPIEL 32.1: Ein Array mit drei Schaltflächen per Code erzeugen

Das Button-Array durchlaufen:

```
        for (int i = 0; i < n; i++)
        {
```

Einen Button "nackt" erzeugen:

```
            buttons[i] = new Button();
```

Eigenschaften zuweisen:

```
            buttons[i].Bounds = new Rectangle(new Point(xpos + i*100, ypos),
                                              new Size(100, 50));
            buttons[i].Text = "Button" + (i+1).ToString();
```

Gemeinsame Ereignisbehandlung anmelden:

```
            buttons[i].Click += new EventHandler(button_Click);
        }
```

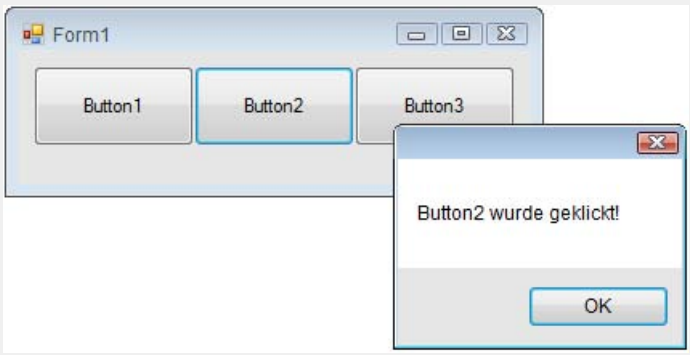
Alle Buttons zum Formular hinzufügen:

```
        this.Controls.AddRange(buttons);
    }
```

Gemeinsamer Eventhandler für *Click*-Ereignis:

```
    private void button_Click(object sender, EventArgs e)
    {
        Button btn = (Button) sender;
        MessageBox.Show(btn.Text + " wurde geklickt!");
    }
}
```

Ergebnis



Alle Buttons sind exakt nebeneinander ausgerichtet. Das Vergrößern des Arrays ist mit keinerlei Mehraufwand an Code verbunden.

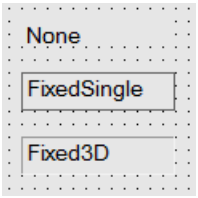
HINWEIS: Die sinnvolle Anwendung eines *CheckBox*-Arrays wird im PB 32.10.2 gezeigt!

32.2 Allgemeine Steuerelemente

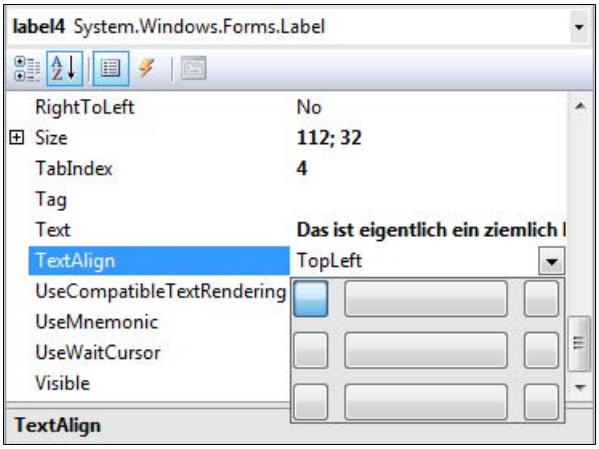
Diese Toolbox-Kategorie enthält standardmäßig die normalerweise am häufigsten benötigten Steuerelemente.

32.2.1 Label

Das harmlose, aber unverzichtbare *Label* dient, im Gegensatz zur *TextBox,* nur zur Anzeige von statischem (unveränderbarem) Text (*Text*-Property). Mit *BorderStyle* haben Sie die Wahl zwischen drei Erscheinungsbildern:

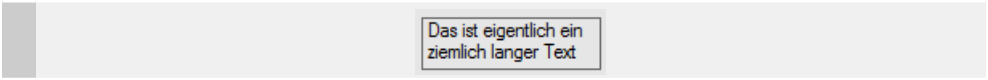


Hervorzuheben ist die *TextAlign*-Eigenschaft, welche die Ausrichtung des Textes auf insgesamt neun verschiedene Arten ermittelt bzw. setzt. Auch hier steht Ihnen ein komfortabler Property-Editor zur Verfügung:



Wenn die *AutoSize*-Eigenschaft auf *True* gesetzt wird, passt sich die Breite des Labels dem aufzunehmenden Text an. Bei *False* ist auch die Anzeige von mehrzeiligem Text möglich.

BEISPIEL 32.2: Der Zeilenumbruch bei *AutoSize = False* erfolgt "intelligent", also beim nächsten passenden Leerzeichen.



HINWEIS: Möchten Sie ein &-Zeichen im *Label* verwenden, können Sie entweder die Eigenschaft *UseMnemonic* auf *False* setzen oder Sie müssen zwei &-Zeichen einfügen.

32.2.2 LinkLabel

Im Grunde handelt es sich bei dieser Komponente um ein *Label*, das mit etwas Funktionalität ergänzt wurde, um einen Hyperlink nachzubilden.

Konzeptionell kann dieses Control leider nur als missglückt angesehen werden. Mit viel Aufwand und Verspieltheit (Sie können mehrere Hyperlinks innerhalb des Controls definieren) wurde am Problem vorbei programmiert. Statt einer simplen *Link*-Eigenschaft, die den Hyperlink enthält und die sich auch zur Entwurfszeit zuweisen lässt, wurde noch eine Klasse integriert, die sich nur zur Laufzeit sinnvoll ansprechen lässt. Zu allem Überfluss muss auch noch die eigentliche Funktionalität, d.h. der Aufruf des Hyperlinks, selbst programmiert werden. Da ist man mit einem einfachen Label fast schneller am Ziel.

Wichtige Eigenschaften

Zur Gestaltung des optischen Erscheinungsbildes können Sie folgende Eigenschaften verwenden:

- *ActiveLinkColor* (der Hyperlink ist aktiv)
- *DisabledLinkColor* (der Hyperlink ist gesperrt)
- *LinkColor* (die Standardfarbe)
- *VisitedLinkColor* (der Hyperlink wurde bereits angeklickt)
- *LinkBehavior* (wie bzw. wann wird der Hyperlink unterstrichen)

Hyperlink einfügen

Verwenden Sie die *Links.Add*-Methode, um zur Laufzeit einen Hyperlink in das Control einzufügen. Übergabewerte sind der Bereich des Hyperlinks bezüglich der *Text*-Eigenschaft und der URL.

BEISPIEL 32.3: Aufruf der MS-Homepage

C# Platzieren Sie einen Hyperlink mit dem Text "Microsoft im Internet" auf dem Formular. Weisen Sie im Formular-Konstruktor den Hyperlink zu:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        linkLabel1.Links.Add(12, 9, "www microsoft.com");
    }
```

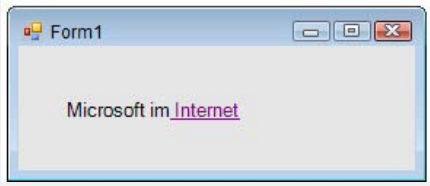
BEISPIEL 32.3: Aufruf der MS-Homepage

Die Zahlenangaben bewirken, dass lediglich das Wort "Internet" als Hyperlink verwendet wird.

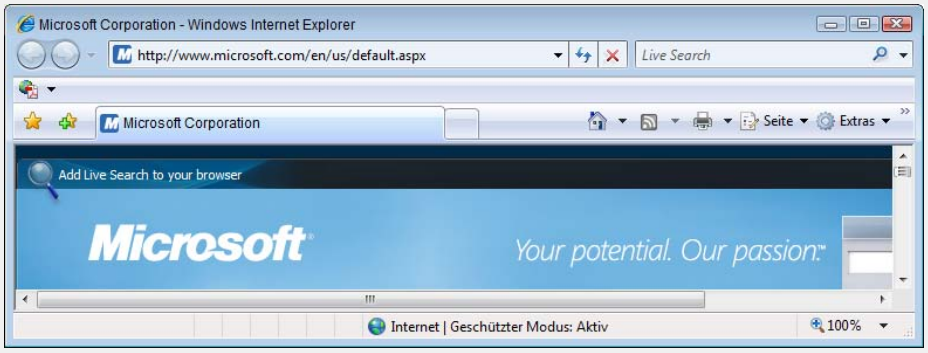
Mit dem Klick auf den *LinkLabel* wird das *LinkClicked*-Ereignis ausgelöst. Hier können Sie der Komponente zum einen mitteilen, dass der Hyperlink besucht wurde, zum anderen sind Sie dafür verantwortlich, den URL auszuwerten (wird im Parameter *e* übergeben) und aufzurufen:

```
    private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
    {
        linkLabel1.Links[linkLabel1.Links.IndexOf(e.Link)].Visited = true;
        System.Diagnostics.Process.Start(e.Link.LinkData.ToString());
    }
}
```

Ergebnis



Wie Sie sehen, erhält der eigentliche Hyperlink auch den Fokus, Sie können also auch mit der *Tabulator*- und der *Enter*-Taste arbeiten.



32.2.3 Button

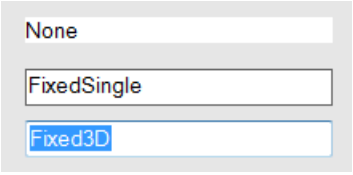
Dieses Steuerelement ist wohl aus (fast) keiner Applikation wegzudenken, über die Funktionalität brauchen wir deshalb kaum Worte zu verlieren. Die folgende Abbildung zeigt vier Vertreter dieser Gattung mit unterschiedlich gesetzter *FlatStyle*-Eigenschaft:

HINWEIS: Entgegen der üblichen Vorgehensweise legen Sie die *Default*-Taste in Dialogboxen nicht mehr über eine Eigenschaft des Buttons, sondern über die *AcceptButton*-Eigenschaft des Formulars fest. Das Gleiche gilt für *CancelButton*.

Mit der *Image*-Eigenschaft bietet sich ein weiteres gestalterisches Mittel, um die trostlosen Buttons etwas aufzupeppen. Alternativ können Sie als Quelle der Grafik auch eine *ImageList* verwenden, in diesem Fall wählen Sie die Grafik mit der *ImageIndex*-Eigenschaft aus.

32.2.4 TextBox

Im Unterschied zum *Label* besteht hier die Möglichkeit, den Text zur Laufzeit zu editieren oder zu markieren. All dies geschieht durch Zugriff auf die *Text*-Eigenschaft. Bei mehrzeiligem Text können Sie über die *Lines*-Eigenschaft auf die einzelnen Textzeilen zugreifen. Das äußere Erscheinungsbild wird, wie beim Label, im Wesentlichen durch die *BorderStyle*-Eigenschaft bestimmt:



Hervorzuheben sind weiterhin folgende Properties:

Eigenschaft	Beschreibung
AutoCompleteSource *AutoCompleteMode* *AutoCompleteCustom-Source*	... ermöglichen automatisches Vervollständigen des einzugebenden Textes
TextAlign	... bestimmt die Ausrichtung des Textes (*Left, Right, Center*).
MaxLength	... legt die maximale Anzahl einzugebender Zeichen fest (Standardeinstellung 32.767 Zeichen).
ReadOnly	... das Control ist schreibgeschützt.

Mehrzeilige Textboxen

Wichtige Eigenschaften in diesem Zusammenhang:

Eigenschaft	Beschreibung
MultiLine	… erlaubt die Eingabe mehrzeiliger Texte (*True*). Für diesen Fall ist auch eine vertikale Scrollbar sinnvoll.
ScrollBars	… bestimmt, ob Bildlaufleisten enthalten sind. Die Eigenschaft zeigt nur bei *MultiLine=True* Wirkung.

Eigenschaft	Beschreibung
AcceptsReturn	Ist diese Eigenschaft *True*, so können Sie mittels Enter-Taste einen Zeilenumbruch einfügen. Ein eventuell vorhandener *AcceptButton* wird damit außer Kraft gesetzt! Bleibt *WantReturns* auf *False,* müssten Sie nach wie vor *Strg+Enter* für einen Zeilenumbruch verwenden.
WordWrap	Damit bestimmen Sie, ob der Text im Eingabefeld am rechten Rand umgebrochen wird (*True*). Der Umbruch wird lediglich auf dem Bildschirm angezeigt, der Text selbst enthält keinerlei Zeilenumbrüche, die nicht eingegeben wurden. Wenn *WordWrap False* ist, entsteht eine neue Zeile nur dort, wo auch ein Zeilenumbruch in den Text eingefügt wurde.
Lines	Zwar gibt es auch eine *Text*-Eigenschaft, doch ist diese für die praktische Arbeit weniger gut geeignet. Sie arbeiten besser mit der *Lines*-Eigenschaft, die einen gezielten Zugriff auf einzelne Zeilen gestattet und die Sie im Stringlisten-Editor oder auch per Quellcode zuweisen können, z.B. Auslesen einer Zeile: `textBox2.Text = textBox1.Lines[3];`

Markieren von Text

SelectionLength, SelectionStart*, SelectedText* gelten für markierte Textausschnitte. *SelectionLength* bestimmt bzw. liefert die Zeichenzahl, *SelectionStart* ermittelt die Anfangsposition, und *SelectedText* setzt bzw. ermittelt den Inhalt.

HINWEIS: Auf diese Properties kann nur zur Laufzeit zugegriffen werden, sie befinden sich **nicht** im Eigenschaftenfenster!

Alternativ können Sie auch die Methoden *SelectAll* bzw. *Select* verwenden.

BEISPIEL 32.4: Wenn man mit der Tab-Taste zur *TextBox* wechselt, wird das erste Zeichen markiert.

C#
```
private void textBox2_Enter1(object sender, EventArgs e);
{
   textBox2.Select(0, 1);
}
```

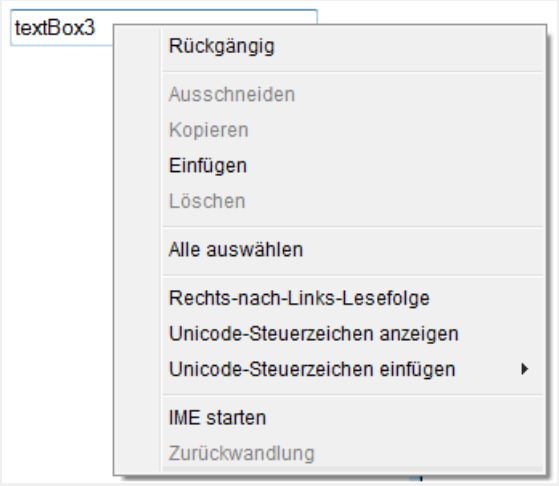
oder

```
   textBox2.SelectionStart  = 0;
   textBox2.SelectionLength = 1;
```

BEISPIEL 32.4: Wenn man mit der Tab-Taste zur *TextBox* wechselt, wird das erste Zeichen markiert.

Ergebnis

Übrigens können Sie zur Laufzeit für jedes Editierfeld ein umfangreiches Kontextmenü aufrufen, über das die wichtigsten Operationen direkt ausführbar sind:



PasswordChar

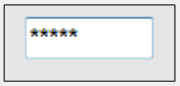
Diese Eigenschaft erlaubt das verdeckte Eingeben eines Passwortes. Sie können das gewünschte Zeichen im Eigenschaften-Fenster oder per Quellcode zuweisen.

BEISPIEL 32.5: Passworteingabe

C#

```
textBox1.PasswordChar = '*';
```

Ergebnis



Automatisches Vervollständigen

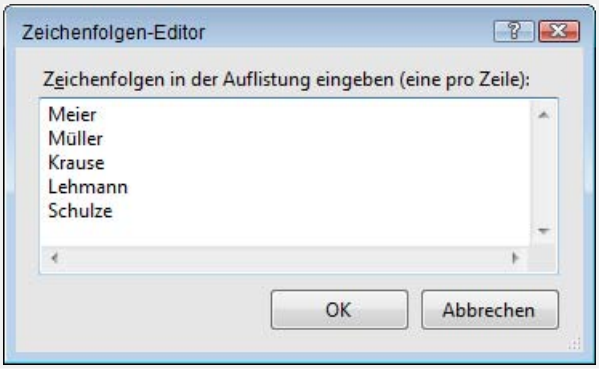
Die Eigenschaften *AutoCompleteCustomSource*, *AutoCompleteSource* und *AutoCompleteMode* einer *TextBox* bewirken das automatische Vervollständigen des eingegebenen Textes durch Vergleich mit einer vorhandenen Liste. Das ist besonders vorteilhaft bei sich häufig wiederholenden Eingabewerten, wie z.B. URLs, Adressen oder Dateinamen.

Voraussetzung für die Verwendung der *AutoCompleteCustomSource*-Eigenschaft ist, dass Sie der *AutoCompleteSource*-Eigenschaft den Wert *CustomSource* zuweisen. Das "Wie" der Textvervollständigung legen Sie mit der *AutoCompleteMode*-Eigenschaft (*None*, *Suggest*, *Append*, *Suggest-Append*) fest.

BEISPIEL 32.6: Automatisches Vervollständigen

Entwurf

Einer *TextBox* (*AutoCompleteSource* = *CustomSource*, *AutoCompleteMode* = *Suggest*) wird die folgende *AutoCompleteCustomSource*-Eigenschaft zugewiesen:



Ergebnis

Nach Eingabe eines Buchstabens öffnet sich eine Liste mit passenden Einträgen (Auswahl über Maus oder Kursortasten):

HINWEIS: *AutoComplete* gibt es auch für die *ComboBox*!

32.2.5 MaskedTextBox

Dieses von der abstrakten *TextBoxBase* abgeleitete Steuerelement benutzt eine Maske die es erlaubt, Benutzereingaben zu filtern.

Von zentraler Bedeutung ist, wen wundert es, die *Mask*-Eigenschaft. Dies ist ein String, der sich aus bestimmten Zeichen zusammensetzt, von denen die wichtigsten in der folgenden Tabelle erklärt werden. Einige Zeichen sind optional, sie können durch Eingabe eines Leerzeichens übergangen werden.

Zeichen	Beschreibung	Zeichen	Beschreibung
0	Ziffer zwischen 0 und 9	.	Dezimal-Trennzeichen
9	Ziffer oder Leerzeichen, optional	,	Tausender-Trennzeichen
#	Ziffer oder Leerzeichen mit Plus(+) oder Minus(-), optional	:	Zeit-Trennzeichen
L	Buchstabe (a-z, A-Z)	/	Datums-Trennzeichen

Zeichen	Beschreibung	Zeichen	Beschreibung
?	Buchstabe (a-z, A-Z), optional	<	konvertiert alle folgenden Zeichen in Kleinbuchstaben
$	Währungssymbol	>	konvertiert alle folgenden Zeichen in Großbuchstaben
&	alphanumerisches Zeichen		

Dezimal-, und Tausender-Trennzeichen, sowie die Symbole für Datum, Zeit und Währung, entsprechen der standardmäßig eingestellten Kultur.

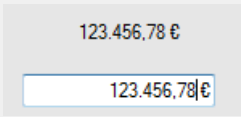
BEISPIEL 32.7: *MaskedTextBox*

C#

Ein Währungswert von 0 bis 999999 wird in eine *MaskedTextBox* eingegeben. Ist die Eingabe komplett, so wird der Inhalt in ein *Label* übernommen. Währungssymbol, Dezimal- und Tausender-Trennzeichen werden zur Laufzeit durch die kulturspezifischen Einstellungen ersetzt.

```
maskedTextBox1.Mask = "999,999.00 $";
...
private void maskedTextBox1_TextChanged(object sender, EventArgs e)
{
   if (maskedTextBox1.MaskCompleted) label1.Text = maskedTextBox1.Text;
}
```

Ergebnis

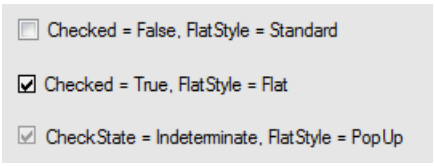


HINWEIS: Eine Leerzeichenkette als Maske wird den vorhandenen Inhalt unverändert belassen, die *MaskedTextBox* benimmt sich dann wie eine einzeilige "normale" *TextBox*.

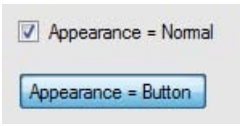
32.2.6 CheckBox

Bei der *CheckBox* entscheidet die *Checked*-Eigenschaft (*True/False*) darüber, ob das Häkchen gesetzt wurde oder nicht. Das äußere Erscheinungsbild kann über die *FlatStyle*-Eigenschaft geringfügig modifiziert werden. Für den Programmierer bietet sich zusätzlich die Möglichkeit, über *CheckState* alle drei Zustände zu bestimmen:

- *Checked*
- *Indeterminate*
- *Unchecked*



Die *Appearance*-Eigenschaft erlaubt es Ihnen, die *CheckBox* optisch in einen Button zu verwandeln, das Verhalten bleibt jedoch gleich:



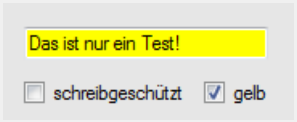
Eine Änderung der *Checked*-Eigenschaft löst das Ereignis *CheckedChanged* aus.

BEISPIEL 32.8: Mit zwei CheckBoxen werden Schreibschutz und Hintergrundfarbe einer *TextBox* eingestellt

C#

```
private void checkBox_CheckedChanged(object sender, EventArgs e)
{
     if (checkBox1.Checked) textBox1.ReadOnly = true; else textBox1.ReadOnly = false;

     if (checkBox2.Checked) textBox1.BackColor = Color.Yellow;
     else textBox1.BackColor = Color.White;
}
```

Ergebnis



Eine weitere interessante Eigenschaft ist *AutoCheck*, ändert man ihren Wert von *True* nach *False*, so wird der Aktivierungszustand der *CheckBox* bei einem Klick nicht mehr automatisch umgeschaltet und der Programmierer muss sich selbst darum kümmern, z.B. durch Auswerten des *Click*-Ereignisses.

BEISPIEL 32.9: *Click*-Eventhandler einer *CheckBox* bei *AutoCheck = False*

C#

```
private void checkBox1_Click(object sender, EventArgs e)
{
    checkBox1.Checked = !checkBox1.Checked;
```

oder allgemeiner:

```
     CheckBox cb = (CheckBox)sender;
     cb.Checked = !cb.Checked;
}
```

32.2.7 RadioButton

Dieses Steuerelement dient zur Auswahl von Optionen innerhalb einer Anwendung. Im Unterschied zur *CheckBox* kann aber innerhalb einer Gruppe immer nur ein einziger *RadioButton* aktiv sein.

Meist fasst man mehrere *RadioButtons* mittels *GroupBox* (oder *Panel*) zu einer Optionsgruppe zusammen. Auch bei dieser Komponente können Sie die *Appearance*-Eigenschaft dazu nutzen, das Erscheinungsbild so zu ändern, dass für jeden *RadioButton* ein *Button* dargestellt wird:

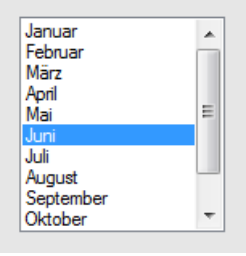
Ausgewertet wird der Status über die *Checked*-Eigenschaft.

BEISPIEL 32.10: Ändern der Hintergrundfarbe des Formulars

C#
```
private void radioButton_CheckedChanged(object sender, EventArgs e)
{
    if (radioButton1.Checked) this.BackColor = Color.White;
    else if (radioButton2.Checked) this.BackColor = Color.Yellow;
    else if (radioButton3.Checked) this.BackColor = Color.Red;
}
```

32.2.8 ListBox

In einer *ListBox* kann eine Auflistung von Einträgen angezeigt werden, von denen der Benutzer mittels Maus oder Tastatur einen oder auch mehrere auswählen kann.



Die wichtigsten Eigenschaften zeigt die folgende Tabelle, auf datenbezogene Eigenschaften wird im Kapitel 35 näher eingegangen.

Eigenschaft	Beschreibung
Items *Items.Count*	… ist die Liste der enthaltenen Einträge, die Sie zur Entwurfszeit auch über den Zeichenfolgen-Editor eingeben können. Über *Items.Count* können Sie die Anzahl bestimmen.
Sorted	… legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*).
SelectedIndex	… setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde).
SelectionMode	… entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist.
SelectedItem	… der Text des ausgewählten Eintrags.

Die Methoden *Items.Add* und *Items.Remove/ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

BEISPIEL 32.11: In die *ListBox* wird zehnmal "Hallo" eingetragen, anschließend wird der zweite Eintrag über seinen Index gelöscht.

C#
```
for (int i= 1; i < 11; i++) listBox1.Items.Add(i.ToString + " Hallo");
listBox1.Items.RemoveAt(1);
```

Häufig will man auch den Inhalt eines (eindimensionalen) Arrays anzeigen.

BEISPIEL 32.12: Der Inhalt eines *String*-Arrays wird in eine *ListBox* übertragen.

C#
```
private void button1_Click(object sender, EventArgs e)
{
    string[] a = {"Sesam", "Weizen", "Hafer", "Gerste"};
    for (int i = 0; i <= a.GetUpperBound(0); i++) listBox1.Items.Add(a[i]);
    listBox1.SelectedIndex = 3;
}
```

Für den Programmierer ist das *SelectedIndexChanged*-Ereignis von besonderem Interesse, da es bei jedem Wechsel zwischen den Einträgen aufgerufen wird.

BEISPIEL 32.13: Nach Auswahl eines Eintrags aus einem Listenfeld wird dieser in eine *Textbox* übernommen.

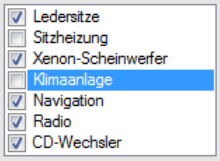
C#
```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e);
{
   textBox1.Text = listBox1.SelectedItem.ToString();
}
```

Ob der Wechsel mittels Maus oder Tastatur erfolgt, ist in diesem Fall egal.

HINWEIS: Ist die Eigenschaft *SelectionMode* auf *MultiSimple* oder *MultiExtended* festgelegt, können Sie auch mehrere Einträge gleichzeitig auswählen. Über *SelectedItems* greifen Sie auf diese Einträge zu.

32.2.9 CheckedListBox

Hierbei handelt es sich um einen nahen Verwandten der "gemeinen" Listbox mit dem Unterschied, dass die einzelnen Einträge gleichzeitig einen *True/False*-Wert (*Checked*) verwalten können.



Für das Setzen der Häkchen per Programm verwenden Sie die *SetItemChecked*-Methode.

BEISPIEL 32.14: Häkchen beim vierten Eintrag setzen

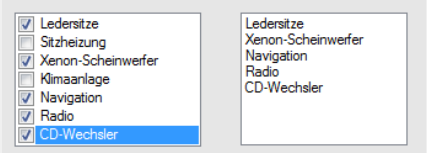
C#
```
checkedListBox1.SetItemChecked(3, true);
```

Über die *CheckedItems*-Collection haben Sie Zugriff auf alle markierten Einträge.

BEISPIEL 32.15: Anzeige aller markierten Einträge in einer weiteren *ListBox*

C#
```
for (int i = 0; i < checkedListBox1.CheckedItems.Count; i++)
    listBox1.Items.Add(checkedListBox1.CheckedItems[i]);
```

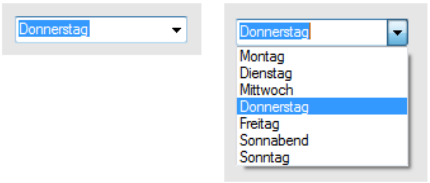
Ergebnis



HINWEIS: Da das Standardverhalten der *CheckedListBox* recht gewöhnungsbedürftig sein dürfte, sollten Sie besser die Eigenschaft *CheckOnClick* auf *True* setzen, damit nicht zweimal geklickt werden muss.

32.2.10 ComboBox

Eine *ComboBox* ist eine Mischung aus Text- und Listenfeld. Sie erlaubt also Eingaben und kann, im Unterschied zum Listenfeld, auch "aufgeklappt" werden:



Hervorzuheben sind folgende Eigenschaften:

Eigenschaft	Beschreibung
DropDownStyle	… *Simple* (nur Texteingabe), *DropDown* (Texteingabe und Listenauswahl, *DropDownList* (nur Listenauswahl).
Items *Items.Count*	… ist die Liste der angezeigten Werte. Über *Items.Count* können Sie die Anzahl bestimmen.
MaxDropDownItems	… die maximale Höhe der aufgeklappten Listbox in Einträgen.
Sorted	… legt fest, ob die Einträge alphabetisch geordnet erscheinen sollen (*True*/*False*).
SelectedIndex	… setzt bzw. ermittelt die Position (Index) des aktuellen Eintrags (-1, wenn nichts ausgewählt wurde).
SelectionMode	… entscheidet, ob Einzel- oder Mehrfachauswahl zulässig ist.
Text	… der Text des ausgewählten Eintrags.

Die Methoden *Items.Add* und *Items.Remove/ItemsRemoveAt* fügen Einträge hinzu bzw. entfernen sie, *Items.Clear* löscht den gesamten Inhalt.

BEISPIEL 32.16: *ComboBox* mit den Namen aller Monate füllen und den ersten Eintrag anzeigen

C#
```
string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
                    "Juli", "August", "September", "Oktober", "November", "Dezember" };
comboBox1.Items.AddRange(monate);
comboBox1.SelectedIndex = 0;          // zeigt "Januar"
```

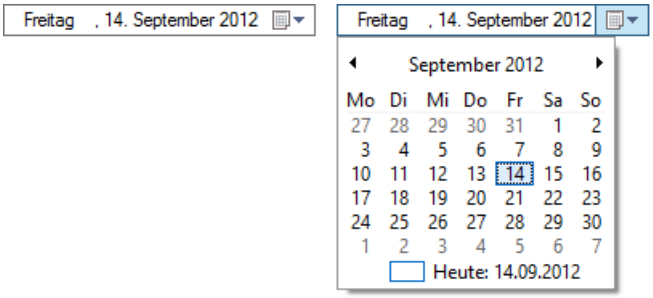
32.2.11 PictureBox

Dieses Control dient der Darstellung von fertigen Grafiken diverser Formate (BMP, GIF, TIFF, PNG …). Außerdem kann mit Grafikmethoden in die *PictureBox* gezeichnet werden, weshalb man sie oft auch als "kleine Schwester" des Formulars bezeichnet. Mittels *Image*-Eigenschaft können Sie diesem Control direkt eine Bildressource zuweisen. Über die *SizeMode*-Eigenschaft steuern Sie, wie die Grafik in den Clientbereich des Controls eingepasst wird bzw. ob sich das Control an die Grafik anpasst.

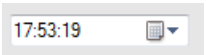
HINWEIS: Mehr zu dieser ebenso interessanten wie leistungsfähigen Komponente erfahren Sie im Kapitel "Grafikprogrammierung" (Abschnitt 33.2.1).

32.2.12 DateTimePicker

Geht es um die platzsparende Auswahl eines Datums- oder eines Zeitwertes, sollten Sie sich mit der *DateTimePicker*-Komponente anfreunden. Diese funktioniert wie eine *ComboBox*, nach dem Aufklappen steht Ihnen ein recht komfortabler Kalender zur Verfügung:



Alternativ kann auch nur eine Uhrzeit bearbeitet werden (Eigenschaft *Format* = *Time*):



Auf alle Möglichkeiten der Konfiguration einzugehen, dürfte den Rahmen dieses Kapitels sprengen. Die wohl wichtigsten Eigenschaften dürften *MinDate*, *MaxDate* für die Beschränkung der Auswahl bzw. *Value* für den Inhalt des Controls sein.

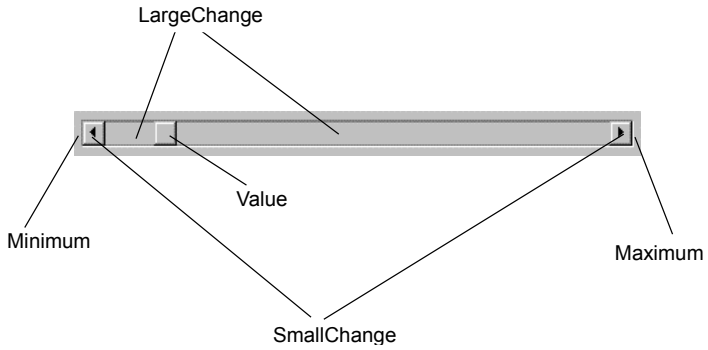
BEISPIEL 32.17: Anzeige des Auswahlwertes

C#
```
private void dateTimePicker1_ValueChanged(object sender, EventArgs e)
{
   MessageBox.Show(dateTimePicker1.Value.ToLongDateString());
}
```

32.2.13 MonthCalendar

Funktionell ist diese Komponente dem *DateTimePicker* sehr ähnlich, der wesentliche Unterschied besteht darin, dass diese Komponente nicht auf- und zugeklappt werden kann und damit natürlich auch mehr Platz auf dem Formular verbraucht:

32.2.14 HScrollBar, VScrollBar



Diese Komponenten werden häufig zum Scrollen von Bild- und Fensterinhalten verwendet.

Maximum, Minimum*, Value*

Sie legen den größten bzw. kleinsten Einstellungswert fest bzw. bestimmen den aktuellen Wert (zwischen *Minimum* und *Maximum)*.

LargeChange, SmallChange

Klickt man neben den "Schieber", so wird der aktuelle Wert (*Value*) um *LargeChange* geändert, beim Klicken auf die Begrenzungspfeile hingegen nur um *SmallChange*.

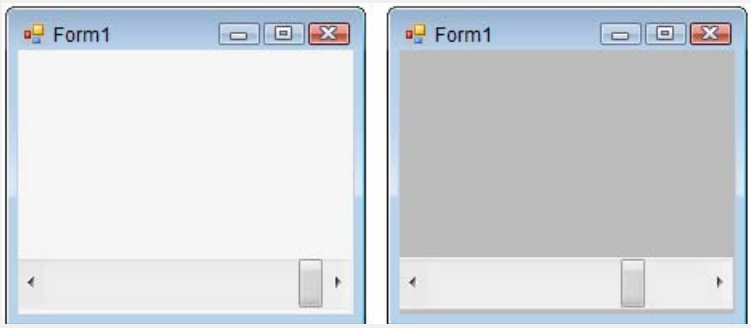
Eine wichtige Rolle spielt auch das *Scroll*-Ereignis, welches immer dann eintritt, wenn die Position des Schiebers verändert wurde (*e.NewValue* enthält den neu gewählten Wert).

BEISPIEL 32.18: Einstellen der Formular-Hintergrundfarbe (Graustufe) mit einer horizontalen Scrollbar (*Minimum = 0, Maximum = 255*)

C#

```
private void hScrollBar1_Scroll(object sender, ScrollEventArgs e)
{
    this.BackColor = Color.FromArgb(255, e.NewValue, e.NewValue, e.NewValue);
}
```

Ergebnis



32.2.15 TrackBar

Neben den Scrollbars bietet sich auch die *TrackBar* für das schnelle Einstellen von Werten an. Hier zwei Varianten:

Die Breite kann bei *AutoSize = False* verändert werden. Ob die *TrackBar* horizontal oder vertikal erscheinen soll, können Sie mittels der *Orientation*-Eigenschaft festlegen. Die Anzahl der Teilstriche ergibt sich aus der Division der *Maximum*-Eigenschaft durch den Wert von *TickFrequency*. Bei der Standardeinstellung *Maximum = 10* und *TickFrequenzy = 1* ergeben sich also 10 Teilstriche.

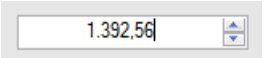
Mit der *TickStyle*-Eigenschaft bestimmen Sie, auf welcher Seite die Teilstriche angezeigt werden sollen. Hier die Konstanten der *TickStyle*-Enumeration:

TickStyle	Erklärung
None	keine Teilstriche
TopLeft	Teilstriche oben (bei horizontaler Ausrichtung) bzw. links (bei vertikaler Ausrichtung)
BottomRight	Teilstriche unten (bei horizontaler Ausrichtung) bzw. rechts (bei vertikaler Ausrichtung (Standardeinstellung)
Both	Teilstriche auf beiden Seiten

Da die übrigen wichtigen Eigenschaften (*Maximum*, *Minimum*, *Value, SmalChange, LargeChange*) sowie Ereignisse (*Scroll*, *ValueChanged*) weitgehend mit denen der Scrollbar übereinstimmen, möchten wir an dieser Stelle auf den vorhergehenden Abschnitt verweisen.

32.2.16 NumericUpDown

Wer hatte nicht schon die Aufgabe, einen Integer- oder Währungswert abzufragen und war an den diversen Fehlermöglichkeiten gescheitert? *NumericUpDown* verspricht Abhilfe. Das Steuerelement entspricht der Kombination einer einzeiligen Textbox mit einer vertikalen Scrollbar.



Angefangen vom Definieren eines zulässigen Bereichs (*Maximum, Minimum*) über die Anzahl der Nachkommastellen (*DecimalPlaces*) bis hin zum Inkrement (*Increment*) bzw. zur Anzeige von Tausender-Trennzeichen (*ThousandsSeparator*) ist alles vorhanden, was das Programmiererherz begehrt. Last but not least, soll die *Value*-Eigenschaft nicht vergessen werden, hier fragen Sie den eingegebenen Wert ab.

HINWEIS: Im Unterschied zu den Steuerelementen *ScrollBar* und *TrackBar* sind die gleichnamigen Eigenschaften von *NumericUpDown* vom Datentyp *decimal* anstatt *int*.

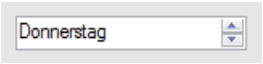
Von besonderem Interesse sind auch hier die Ereignisse *Scroll* (beim Klicken auf eine der beiden Schaltflächen) und *ValueChanged* (beim Ändern des Werts).

BEISPIEL 32.19: Meldung beim Überschreiten eines Limits

C#
```
private void numericUpDown1_ValueChanged(object sender, EventArgs e)
{
    if (numericUpDown1.Value > 1000) MessageBox.Show("Wertebereichsüberschreitung!");
}
```

32.2.17 DomainUpDown

Für die Auswahl von Werten aus einer vorgegebenen Liste können Sie anstatt einer *ListBox* oder einer *ComboBox* auch eine *DomainUpDown*-Komponente verwenden. Dieses Control kombiniert die Features eines *NumericUpDown*-Steuerelements mit denen einer *ComboBox* (lässt sich allerdings nicht aufklappen). Rein äußerlich scheint es zunächst keinen Unterschied zur *NumericUpDown* zu geben, die Funktionalität ist jedoch grundsätzlich verschieden, bestehen doch die Einträge nicht aus Zahlen sondern aus Strings (Reihenfolge kann mittels *Sort*-Eigenschaft alphabetisch sortiert werden).



Die zulässigen Auswahlwerte übergeben Sie der Komponente in der Collection *Items*, den gewählten Wert ermitteln Sie mit der *Text*-Eigenschaft oder Sie können auch den Listenindex mit *SelectedIndex* abrufen.

BEISPIEL 32.20: Der erste Listeneintrag wird angezeigt

C#
```
domainUpDown1.SelectedIndex = 0;
```

32.2.18 ProgressBar

Ist Ihr Computer bei manchen Aufgaben nicht schnell genug, ist es sinnvoll, dem Nutzer ein Lebenszeichen zum Beispiel in Gestalt eines Fortschrittbalkens zu geben. Genau diese Aufgabe übernimmt die *ProgressBar*:

Die Eigenschaften *Minimum*, *Maximum* und *Value* sind vergleichbar mit denen der *ScrollBar*. Sie können *Value* einen Wert zuweisen, andererseits aber auch die Methode *PerformStep()* aufrufen.

BEISPIEL 32.21: Bei jedem Klick auf den *Button* bewegt sich der Balken weiter und hat nach zehnmaligen Klicken das Maximum erreicht.

C#
```
private void button1_Click(object sender, EventArgs e)
{
    progressBar1.Maximum = 100; progressBar1.Step = 10;
    progressBar1.PerformStep();
}
```

HINWEIS: Mit der *Style*-Eigenschaft (*Blocks*, *Continuous*, *Marquee*) können Sie das Aussehen des Balkens verändern.

32.2.19 RichTextBox

Wollten Sie nicht schon immer einmal Ihre Anwendungen mit einem kleinen Texteditor vervollständigen, der auch verschiedene Schriftarten, Schrift- und Absatzformate sowie Grafiken zulässt? Dann sollten Sie sich unbedingt das *RichTextBox*-Control ansehen.

HINWEIS: Mit RTF ist das "Rich Text"-Format gemeint, das als Austauschformat zwischen verschiedenen Textverarbeitungsprogrammen fungiert.

Viele Eigenschaften entsprechen denen der *TextBox*, so ist der gesamte Inhalt in der *Text*-Eigenschaft enthalten. Mit diversen *Selection...* -Eigenschaften lassen sich die markierten Textabschnitte formatieren.

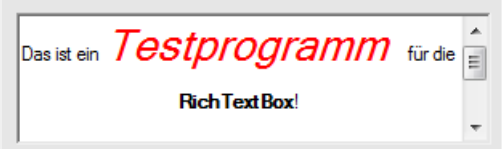
BEISPIEL 32.22: Selektierten Text in Fett- und Kursivschrift formatieren

C#
```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
                                      richTextBox1.SelectionFont.Style ^ FontStyle.Bold);
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont,
                                      richTextBox1.SelectionFont.Style ^ FontStyle.Italic);
```

BEISPIEL 32.23: Schriftgröße und -farbe ändern, Kursivschrift und markierte Zeile zentrieren

C#
```
richTextBox1.SelectionFont = new Font(richTextBox1.SelectionFont.Name, 20f);
richTextBox1.SelectionColor = Color.Red;
richTextBox1.SelectionAlignment = HorizontalAlignment.Center;
```

Ergebnis



Zum Laden bzw. Abspeichern von RTF-Dateien stehen die Methoden *LoadFile* und *SaveFile* zur Verfügung.

BEISPIEL 32.24: Laden der Datei *test.rtf* aus dem Anwendungsverzeichnis

C#
```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        richTextBox1.LoadFile(Application.StartupPath + "\\test.rtf");
    }
    catch (System.IO.IOException ioe)
    {
         MessageBox.Show(ioe.Message);
    }
}
```

Bei der *RichTextBox* handelt es sich nur auf den ersten Blick um einen kompletten Texteditor, denn Sie haben sicherlich bereits festgestellt, dass Sie ja auch noch ein Menü bzw. ein Toolbar benötigen.

HINWEIS: Eine komplette kleine Textverarbeitung auf Basis der *RichTextBox* wird in unserem [Visual C# 2012 Kochbuch] beschrieben.

32.2.20 ListView

Die *ListView* ist eine ziemlich komplexe Komponente, sie ermöglicht Ihnen die Anzeige einer Liste mit Einträgen, die optional auch mit einem Icon zwecks Identifikation des Typs ausgestattet werden können, wozu zusätzlich eine oder mehrere *ImageList*-Komponenten erforderlich sind. Außerdem kann zu jedem Eintrag eine kleine Checkbox hinzugefügt werden, wodurch sich eine bequeme und übersichtliche Auswahlmöglichkeit ergibt.

Die Einsatzgebiete einer *ListView* sind äußerst vielgestaltig, z.B. Darstellen von Datenbankinhalten oder Textdateien. Außerdem kann die Komponente auch Nutzereingaben entgegennehmen, z.B. Dateiauswahl. Als Windows-Nutzer haben Sie garantiert schon mit diesem Control gearbeitet, jeder einzelne Ordner auf dem Desktop ist im Grunde ein *ListView*-Objekt und fungiert als eine Art Container für eine Anzahl von einzelnen Items, die über Grafik und Text verfügen können.

ListViewItem

Das *ListViewItem*-Objekt repräsentiert einen einzelnen Eintrag (Item) im *ListView*. Jedes *Item* kann mehrere *Subitems* haben, die zusätzliche Informationen bereitstellen.

Die Anzeige der Items ist auf vier verschiedene Arten möglich:

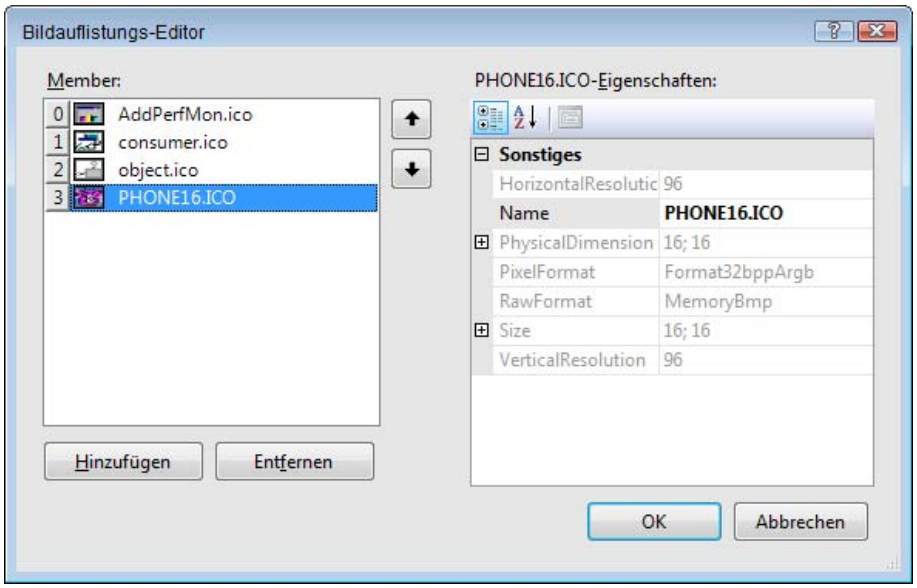
- mit großen Icons
- mit kleinen Icons

- mit kleinen Icons in einer vertikalen Liste
- Gitterdarstellung mit Spalten für Untereinträge (Detailansicht)

Die *ListView*-Komponente erlaubt einfache oder mehrfache Selektion, letztere funktioniert ähnlich wie bei einer *ListBox*-Komponente.

ImageList

Meist wird die *ListView* zusammen mit einer (oder auch mehreren) *ImageList*-Komponenten eingesetzt, welche die Bilddaten speichern. Die Zuweisung erfolgt über die Eigenschaften *LargeImageList* bzw. *SmallImageList.* Jede *ImageList*-Komponenten hat eine *Images*-Auflistung, die Sie über das Eigenschaften-Fenster (F4) erreichen und die Sie mit Hilfe des *Image-Auflistungs-Editor*s mit diversen Icons füllen können.



Übersichten zur ListView

Eigenschaft	Bedeutung
View	... ermöglicht die Einstellung des Anzeigemodus (siehe oben): *List, SmallIcon, LargeIcon, Details*
LargeImageList *SmallImageList* *StateImageList*	Auswahl der *ImageList*-Objekte, welche die Bilddateien für große (32 x 32), kleine (16 x 16) Icons bzw. für die Darstellung der Checkbox bereitstellen (nur für *CheckBoxes* = *True*).
CheckBoxes	... Ein- bzw. Ausblenden der Checkbox (*True/False*)
CheckItems	... Zugriff auf die *CheckedListViewItemCollection*, um die aktivierten Einträge festzustellen

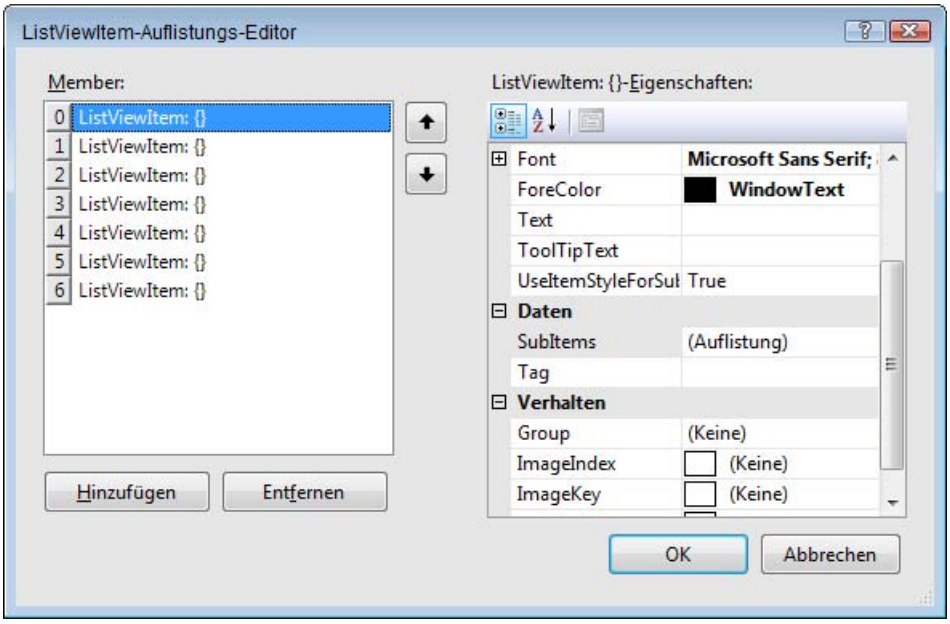
Eigenschaft	Bedeutung
Columns	... Zugriff auf die *ColumnHeaderCollection* (nur für *View = View.Details*)
Items	... Hinzufügen/Entfernen von Einträgen durch Zugriff auf die *ListView-ItemCollection*, die entsprechende Methoden bereitstellt
ImageIndex	... der Bildindex
LabelEdit	... erlaubt/verbietet das Editieren von Einträgen (*True*)
Sorting	... alphabetisches Sortieren der Einträge (*None*, *Ascending*, *Descending*)
AllowColumnReorder	nur für Detailansicht (*View = View.Details*): ... erlaubt Vertauschen der Spalten zur Laufzeit durch Anfassen mit der Maus (*True*)
FullRowSelect	... komplette Zeile wird selektiert (*True*)
GridLines	... Anzeige von Gitterlinien (*True*)
HeaderStyle	... versteckt Spaltentitel (*None*), zeigt ihn an (*NoneClickable*) bzw. Spaltentitel funktioniert wie ein Button (*Clickable*)
HideSelection	... selektierte Einträge bleiben markiert, wenn die *ListView* den Fokus verliert (*False*)

Methode	Beschreibung
BeginUpdate *EndUpdate*	... erlaubt das Hinzufügen von mehreren Items, ohne dass *ListView* jedes Mal neu gezeichnet wird
Clone	... überträgt Items in andere *ListView*
GetItemAt	... ermöglicht das Bestimmen des *Items* durch Klick mit der Maus auf ein *SubItem* (*View = View.Details*)
EnsureVisible	... ermöglicht die Anzeige eines gewünschten Eintrags nach Neuaufbau

Ereignis	... wird ausgelöst
BeforeLabelEdit *AfterLabelEdit*	... bevor bzw. nachdem ein Eintrag editiert wurde (nur bei *LabelEdit = True*), kann zur Gültigkeitsüberprüfung benutzt werden
ItemActivate	... wenn ein bestimmtes Item ausgewählt wurde, kann zum Auslösen von Aktionen benutzt werden
ColumnClick	... wenn einen Spaltentitel geklickt wurde, kann z.B. zum Sortieren verwendet werden
ItemCheck	... wenn auf eine *CheckBox* geklickt wurde (nur für *CheckBoxes = True*)

Die Vorgehensweise scheint recht einfach zu sein:

- Verknüpfen Sie die *ListView* mit einer *ImageList* mit großen Icons (z.B. 32 x 32) per *LargeImageList*-Eigenschaft und mit einer weiteren *ImageList* die kleinen Icons (16 x 16) per *SmallImageList*-Eigenschaft.
- Neue Einträge erstellen Sie zur Entwurfszeit über die *Items*-Eigenschaft, die auch einen eigenen Editor bereitstellt. Wie Sie der folgenden Abbildung entnehmen können, verfügt jeder Eintrag wiederum über eine Auflistung *SubItems*, die den einzelnen Spalten in der Detailansicht entsprechen.



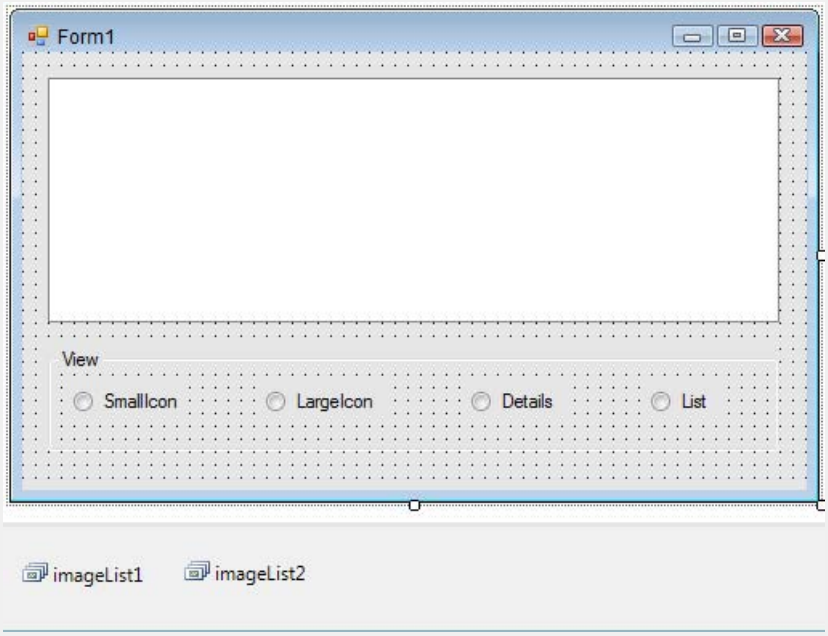
- Wichtig sind vor allem die Eigenschaften *ImageIndex* und *Text*, da beide das Aussehen des *Items* beeinflussen.
- Möchten Sie in der Detailansicht weitere Daten (Spalten) anzeigen, können Sie diese mit der *Columns*-Eigenschaft hinzufügen. Eingetragen werden später die *Text*-Eigenschaften der *SubItems*.
- Die Art der Anzeige legen Sie mit der *View*-Eigenschaft der *ListView* fest (Liste, Details etc.).

BEISPIEL 32.25: Stundenplan

C#

Es soll ein einfacher "Stundenplan" erstellt werden. Die folgende Abbildung gibt einen Überblick über die Anordnung der Komponenten. Wichtig sind die *ListView*-Komponente (oben) und die beiden *ImageList*-Komponenten, die automatisch im Komponentenfach abgelegt werden.

BEISPIEL 32.25: Stundenplan



Füllen Sie mit dem Image-Auflistungs-Editor (siehe oben) die erste *ImageList* mit "großen" und die zweite *ImageList* mit "kleinen" Icons. Weisen Sie der *LargeImageList*- und der *SmallImageList*-Eigenschaft der *ListView* die beiden *ImageList*-Komponenten zu.

Beim Laden von *Form1* wird dem *Click*-Ereignis der vier *RadioButton*s ein gemeinsamer Event-Handler zugewiesen:

```
private void Form1_Load(object sender, EventArgs e)
{
    radioButton1.Click += new EventHandler(this.commonClick);
    radioButton2.Click += new EventHandler(this.commonClick);
    radioButton3.Click += new EventHandler(this.commonClick);
    radioButton4.Click += new EventHandler(this.commonClick);
}
```

Der gemeinsame Event-Handler:

```
private void commonClick(object sender, System.EventArgs e)
{
    showListView();
}
```

Die Anzeige der *ListView*:

```
private void showListView()
{
    listView1.Items.Clear();
```

BEISPIEL 32.25: Stundenplan

```
    if (radioButton1.Checked) listView1.View = View.SmallIcon;
    if (radioButton2.Checked) listView1.View = View.LargeIcon;
    if (radioButton3.Checked) listView1.View = View.Details;
    if (radioButton4.Checked) listView1.View = View.List;
```

Spalten für *Items* and *SubItems* definieren:

```
    listView1.Columns.Add("", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Montag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Dienstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Mittwoch", -2, HorizontalAlignment.Center);
    listView1.Columns.Add("Donnerstag", -2, HorizontalAlignment.Left);
    listView1.Columns.Add("Freitag", -2, HorizontalAlignment.Center);
    listView1.LabelEdit = true;           // Editieren erlauben:
    listView1.AllowColumnReorder = true;  // Ändern der Spaltenanordnung erlauben
    listView1.CheckBoxes = true;
    listView1.FullRowSelect = true;
    listView1.GridLines = true;
    listView1.Sorting = SortOrder.Ascending;
```

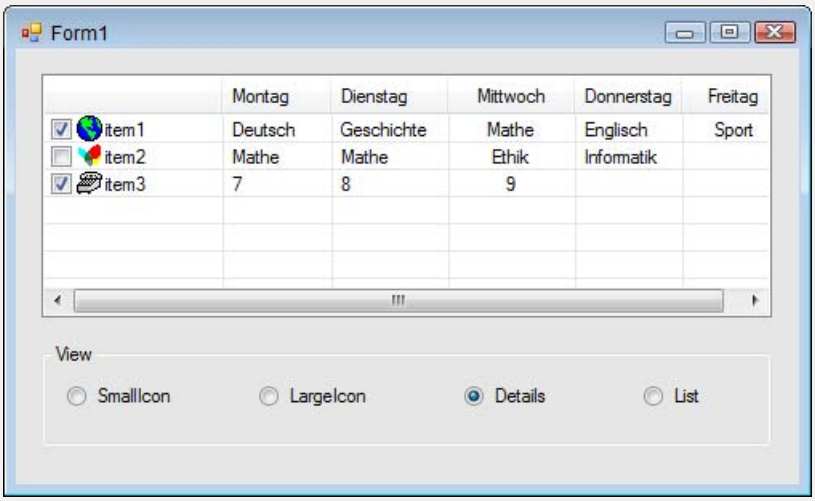
Schließlich drei *Items* mit Gruppen von *SubItems* erzeugen und zur *ListView* hinzufügen:

```
    ListViewItem item1 = new ListViewItem("item1", 0);
    item1.Checked = true;
    item1.SubItems.Add("Deutsch");
    item1.SubItems.Add("Geschichte");
    item1.SubItems.Add("Mathe");
    item1.SubItems.Add("Englisch");
    item1.SubItems.Add("Sport");
    listView1.Items.Add(item1);

    ListViewItem item2 = new ListViewItem("item2", 1);
    item1.Checked = true;
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Mathe");
    item2.SubItems.Add("Ethik");
    item2.SubItems.Add("Informatik");
    item2.SubItems.Add("");
    listView1.Items.Add(item2);

    ListViewItem item3 = new ListViewItem("item3", 2);
    item3.Checked = true;
    item3.SubItems.Add("7");
    item3.SubItems.Add("8");
    item3.SubItems.Add("9");
    listView1.Items.Add(item3);
  }
```

BEISPIEL 32.25: Stundenplan

Ergebnis

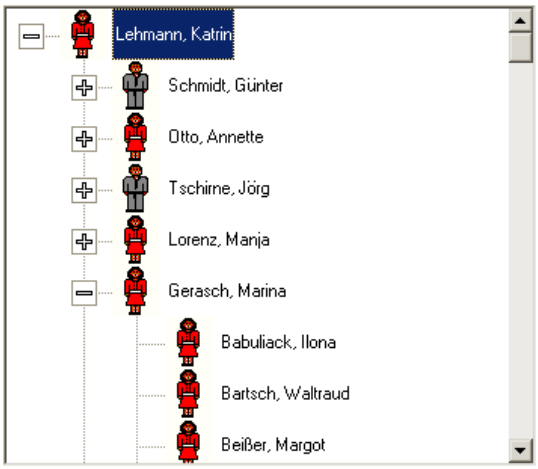
Starten Sie das Programm und experimentieren Sie nun mit den verschiedenen Ansichten. Die Abbildung zeigt die Detailansicht:



HINWEIS: Leider können wir an dieser Stelle nicht auf alle Aspekte der *ListView*-Programmierung eingehen, die sich nur anhand von mehreren Beispielen plausibel erklären lässt, siehe [Visual C# 2012 Kochbuch].

32.2.21 TreeView

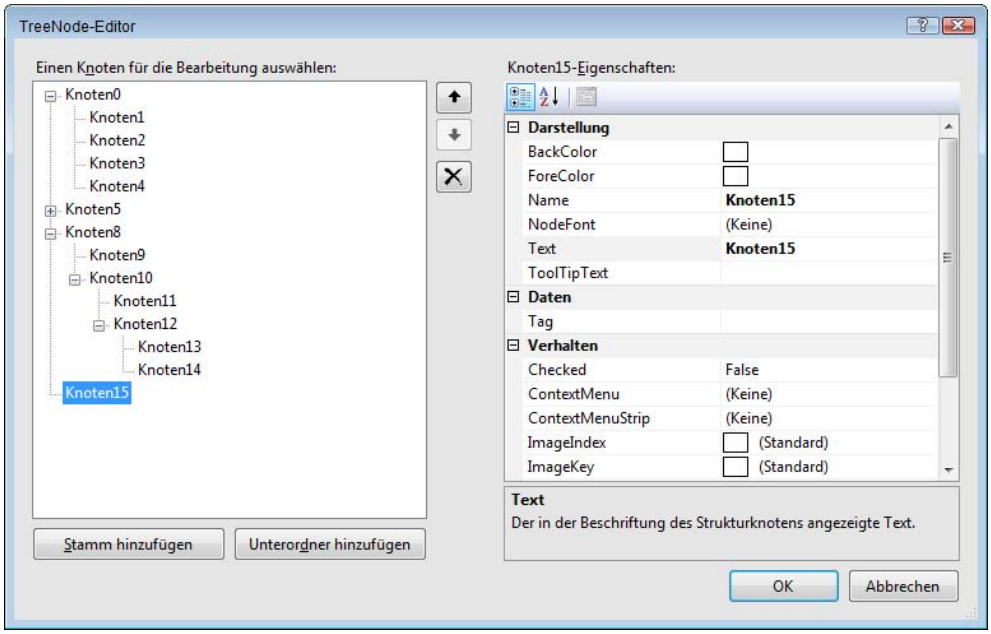
Wollen Sie den Anwendern Ihrer Programme mehr bieten als nur langweilige Listendarstellungen und primitive Eingabemasken? Oder müssen Sie hierarchische Abhängigkeiten grafisch darstellen?



Falls ja, dann kommen Sie wohl kaum um die Verwendung der *TreeView*-Komponente herum. Ganz abgesehen davon, dass hier eine Baumdarstellung wesentlich übersichtlicher ist als eine Tabelle, können Sie durch den Einsatz grafischer Elemente auch noch reichlich Eindruck schinden.

Wie auch die *ListView*-Komponente dient die *TreeView* "lediglich" als Container für eine Liste von *TreeNodes* (Knoten), die jedoch im Gegensatz zur *ListView* hierarchisch angeordnet sind. D h., ausgehend von einem Knoten werden weitere Unterknoten angeordnet usw.

Zur Entwurfszeit können Sie diese Knoten mit einem eigenen Editor (Eigenschaft *Nodes*) hinzufügen (ähnlich dem Erzeugen von Verzeichnissen auf einer Festplatte):



Knoten zur Laufzeit erzeugen

Zur Laufzeit gestaltet sich dieses Vorgehen schon etwas aufwändiger und erfordert vom Programmierer einiges Vorstellungsvermögen, gilt es doch, sich mit Hilfe von Collections und Methoden buchstäblich durch den Baum zu "hangeln".

BEISPIEL 32.26: Dynamisches Füllen einer *TreeView*

C#

Zunächst den Baum löschen und die automatische Aktualisierung ausschalten (bessere Performance):

```
treeView1.BeginUpdate;
treeView1.Nodes.Clear;
```

Wir fügen den ersten (Root-)Knoten bzw. Stammelement ein:

```
treeView1.Nodes.Add("Deutschland");
```

BEISPIEL 32.26: Dynamisches Füllen einer *TreeView*

Da der erste Knoten den Index 0 in der *Nodes*-Collection erhält, können wir direkt über den Index auf diesen Knoten zugreifen und weitere untergeordnete Knoten erzeugen, die zur *Nodes*-Auflistung des Knotens hinzugefügt werden:

```
treeView1.Nodes[0].Nodes.Add("Bayern");
treeView1.Nodes[0].Nodes.Add("Sachsen");
treeView1.Nodes[0].Nodes.Add("Thüringen");
```

Das gleiche Prinzip auf die nächstfolgende Knotenebene angewendet:

```
treeView1.Nodes[0].Nodes[0].Nodes.Add("München");
treeView1.Nodes[0].Nodes[0].Nodes.Add("Nürnberg");
```

Doch eigentlich wird jetzt die Schreiberei etwas zu aufwändig. Bequemer ist die folgende Variante, bei der ein *TreeNode*-Objekt von der *Add*-Methode zurückgegeben wird:

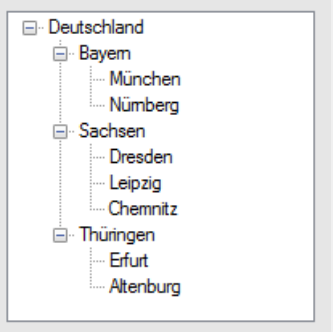
```
TreeNode n = treeView1.Nodes[0].Nodes[1];
n.Nodes.Add("Dresden");
n.Nodes.Add("Leipzig");
n.Nodes.Add("Chemnitz");
```

Es genügt also, dass wir uns auf einen bestimmten Knoten beziehen, um weitere Unterknoten zu erzeugen.

Zum Schluss noch die Aktualisierung einschalten, sonst ist nichts zu sehen:

```
treeView1.EndUpdate();
```

Ergebnis

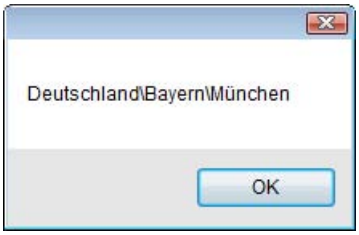


Auswerten des aktiven Knotens

Über das *AfterSelect*-Ereignis können Sie auswerten, welcher Knoten gerade aktiviert wurde. Der Knoten selbst wird als *e.Node*-Objekt als Parameter übergeben:

```
private void treeView1_AfterSelect(object sender, TreeViewEventArgs e)
{
    MessageBox.Show(e.Node.Text);
    MessageBox.Show(e.Node.FullPath);
}
```

Während *Node.Text* lediglich die Beschriftung des aktuellen Knotens zurückgibt, enthält *Node.FullPath* den kompletten Pfad bis zur Root:



Wichtige Eigenschaften von TreeView

Die *TreeView* ist ein hochkomplexes Steuerelement und verfügt über eine Flut von Eigenschaften, Methoden und Ereignissen, bei denen man schnell die Übersicht verlieren kann. Gerade deshalb ist es sinnvoll, wenn man wenigstens die wichtigsten Klassenmitglieder kennt. Dabei müssen wir natürlich auch die untergeordnete Klasse *TreeNode*, welche einen einzelnen Knoten innerhalb von *TreeView* repräsentiert, mit einbeziehen.

HINWEIS: Wer sich eine einfache aber trotzdem sinnvolle *TreeView*-Anwendung anschauen möchte, den verweisen wir auf das Praxisbeispiel 32.10.3 am Ende des Kapitels.

Eigenschaft	Beschreibung
CheckBoxes	... legt fest, ob CheckBoxen angezeigt werden (*True*/***False***)
FullRowSelect	... legt fest, ob die gesamte Breite farblich hervorgehoben wird (*True*/***False***)
HideSelection	... entfernt farbliche Hervorhebung des selektierten Knotens (***True***/*False*)
HotTracking	... legt fest, ob Knotenbezeichnung als Hyperlink erscheint (*True*/***False***)
ImageIndex	Index des Bildchens welches erscheint, wenn Knoten nicht selektiert ist
ImageList	... verweist auf die zugeordnete *ImageList*.
Indent	... Einzugsbreite der untergeordneten Knoten (Pixel)
LabelEdit	... legt fest, ob Anwender Knotenbeschriftung editieren kann (*True*/***False***)
Nodes	... Auflistung mit allen *TreeNode*-Objekten der untergeordneten Ebene
SelectedImageIndex	... Index des Bildchens, welches bei selektiertem Knoten angezeigt wird
SelectedNode	... liefert selektiertes *TreeNode*-Objekt
ShowLines	... gibt an, ob Linien zwischen den Knoten gezeichnet werden sollen (***True***/*False*)

Wichtige Methoden von TreeView

Methode	Beschreibung
BeginUpdate()	... deaktiviert das Neuzeichnen des Controls
CollapseAll()	... reduziert alle Knoten
EndUpdate()	... aktiviert das Neuzeichnen des Controls
ExpandAll()	... expandiert alle Knoten
Sort()	... sortiert alle Knoten

Wichtige Ereignisse von TreeView

Beim Aufklappen und Schließen der Knoten löst *TreeView* Ereignispärchen aus.

Ereignisse	Ereignis wird ausgelöst,
BeforeExpand *AfterExpand*	... bevor/nachdem Knoten geöffnet worden ist.
BeforeSelect *AfterSelect*	... bevor/nachdem Knoten ausgewählt wurde.
BeforeCollaps *AfterCollaps*	... bevor/nachdem Knoten geschlossen wurde.
NodeMouseClick	... wenn Benutzer auf einen Knoten klickt.

Wichtige Eigenschaften von TreeNode

Eigenschaft	Beschreibung
FirstNode	Der erste untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens.
Index	Position des aktuellen Knotens in der *Nodes*-Auflistung des übergeordneten Knotens
IsExpanded	*True*, falls *TreeNode*-Objekt expandiert ist.
IsSelected	*True*, falls *TreeNode*-Objekt selektiert ist.
LastNode	Das letzte untergeordnete *TreeNode*-Objekt in der *Nodes*-Auflistung des aktuellen Knotens.
NextNode	Das nächste gleichrangige *TreeNode*-Objekt
Nodes	Auflistung, die alle *TreeNode*-Objekte der untergeordneten Ebene enthält.
Parent	Das übergeordnete *TreeNode*-Objekt
PrevNode	Das vorhergehende gleichrangige *TreeNode*-Objekt

Wichtige Methoden von TreeNode

Methode	Beschreibung
Collapse()	Das *TreeNode*-Objekt wird reduziert.
Expand()	Das *TreeNode*-Objekt wird expandiert.
ExpandAll()	Alle untergeordneten *TreeNode*-Objekte werden expandiert.
Toggle()	*TreeNode* wechselt zwischen reduziertem und erweitertem Zustand.

32.2.22 WebBrowser

Seit der Einführung von Windows Forms 2.0 steht dieses hochkomplexe Steuerelement zur Verfügung (es handelt sich um eine verwaltete Wrapper-Klasse für das bekannte Internet Explorer ActiveX Control).

Die Programmierung eines voll funktionsfähigen Internetbrowsers erfordert erstaunlich wenig Aufwand.

BEISPIEL 32.27: *WebBrowser*

C#

Nachdem Sie das *WebBrowser*-Control von der Toolbox auf das Formular gezogen haben (*Dock = Bottom*), brauchen Sie nur noch eine *TextBox* für die URL und einen *Button* mit folgendem Ereigniscode hinzuzufügen:

```
private void button1_Click(object sender, EventArgs e)
{
      Uri aUri = new Uri(textBox1.Text);
      webBrowser1.Url = aUri;
}
```

Ergebnis

Die *Url*-Eigenschaft des *WebBrowser*-Controls ist vom Typ *Uri*. Diese Klasse repräsentiert einen *Uniform Resource Identifier* (URI) und kapselt den Zugriff auf die Bestandteile eines URI. Weitere wichtige Eigenschaften sind *IsWebWebBrowserContextMenuEnabled*, *SecurityLevel* und *WebBrowserShortcutEnabled*.

HINWEIS: Weitere Funktionen, wie die Aktualisierung der Anzeige etc., sind über das Kontextmenü des Webbrowsers verfügbar.

32.3 Container

Mit diesen Komponenten können Sie ein übersichtliches und anpassungsfähiges Outfit der Bedienoberfläche Ihrer Anwendung erreichen.

32.3.1 FlowLayout/TableLayout/SplitContainer

Diese schon mit .NET 2.0 eingeführten Komponenten ermöglichen ein flexibles Formularlayout und wurden bereits im Abschnitt 31.2.6 des Vorgängerkapitels näher beschrieben.

32.3.2 Panel

Das auf den ersten Blick etwas unscheinbar wirkende Panel dürfte eines der wichtigsten Gestaltungsmittel für Dialog-Oberflächen sein. Die wichtigste Fähigkeit dieser Komponente: sie kann weitere Komponenten in ihrem Clientbereich aufnehmen. Damit verhält sie sich fast wie ein Formular, es gibt eine *Controls*-Auflistung und sogar eine *AutoScroll*-Eigenschaft ist vorhanden!

Wie auch bei der *GroupBox* gilt:

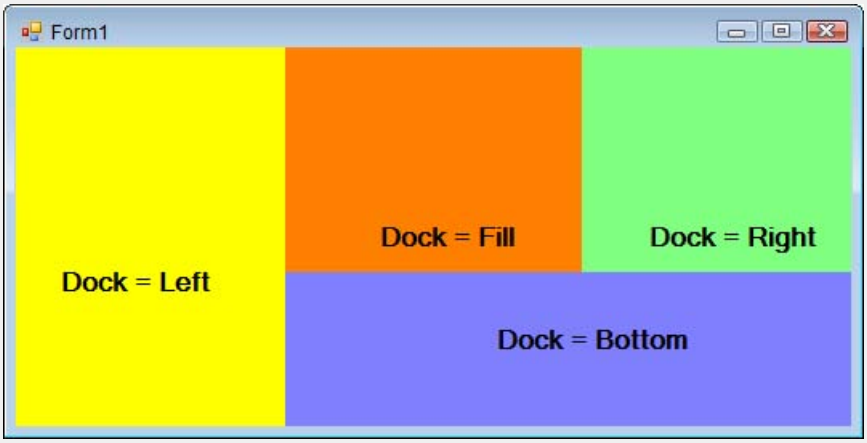
HINWEIS: Beachten Sie beim Entwurf, dass zuerst das *Panel* angelegt werden muss und erst dann Steuerelemente innerhalb desselben abgelegt werden können.

Oberflächen gestalten

Im Zusammenhang mit der *Dock*- bzw. *Anchor*-Eigenschaft bieten sich fast unbegrenzte Möglichkeiten, den Clientbereich des Formulars in einzelne Arbeitsbereiche aufzuteilen und dynamisch auf Größenänderungen zu reagieren.

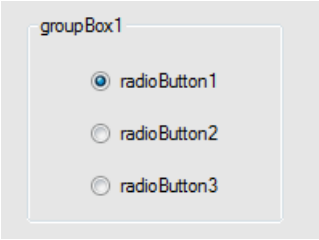
HINWEIS: Nutzen Sie die *SplitContainer*-Komponente, kann der Nutzer zur Laufzeit die Größe der Panels ändern.

BEISPIEL 32.28: Mehrere mit Dock ausgerichtete *Panels*



32.3.3 GroupBox

Wenn man mehrere Steuerelemente mit einer *GroupBox* umgibt, können diese zu einer Einheit zusammengefasst werden. Insbesondere beim Gruppieren von *RadioButtons* dürfte die *GroupBox* die erste Wahl sein.



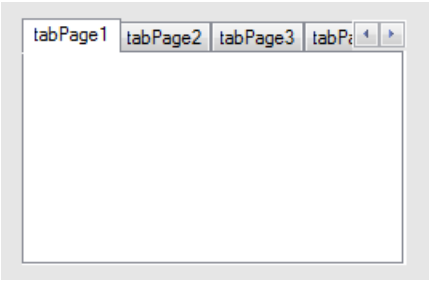
Wie auch das *Panel* verfügt die *GroupBox* über eine eigene *Controls*-Auflistung, zu der weitere Elemente mit *Add* bzw. *AddRange* hinzugefügt werden können.

Eine *GroupBox* reiht sich zwar mit ihrer *TabIndex*-Eigenschaft in die Tabulator-Reihenfolge der übrigen Steuerelemente des Formulars ein, für die in der *GroupBox* enthaltenen Elemente gibt es jedoch eine eigene Reihenfolge, die mit *TabIndex = 0* beginnt.

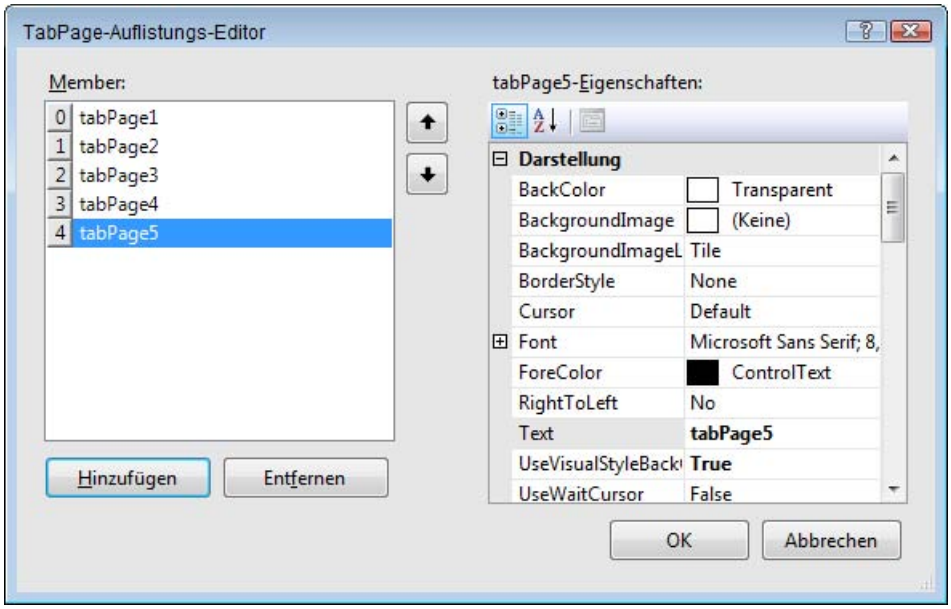
HINWEIS: Beachten Sie beim Entwurf, dass zuerst die *GroupBox* angelegt werden muss und erst dann Steuerelemente innerhalb derselben abgelegt werden können.

32.3.4 TabControl

Diese Komponente bietet die Funktionalität eines *Panel*s und die Möglichkeit, über ein Register auf weitere Seiten zuzugreifen:



Für das Zuweisen der Eigenschaften und für das Hinzufügen neuer Registerblätter steht Ihnen der "TabPage-Auflistungs-Editor" zur Verfügung (erreichbar über die Eigenschaft *TabPages*):



Um per Code neue Registerkarten (*TabPage*-Objekte) hinzuzufügen, verwenden Sie (wie könnte es bei einem Container-Control auch anders sein) die *Add*-Methode der *Controls*-Auflistung des *TabControl*-Objekts.

BEISPIEL 32.29: Eine Registerkarte mit der Beschriftung "XYZ" wird zu einem *TabControl* hinzugefügt

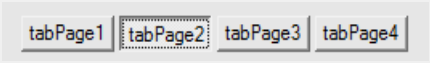
C#
```
TabPage tabPg = new TabPage("XYZ");
tabControl1.Controls.Add(tabPg);
```

Wichtige Eigenschaften

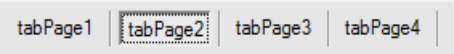
Eigenschaft	Beschreibung
Alignment	... bestimmt Anordnung der Karteireiter
Appearance	... bestimmt Erscheinungsbild des *TabControl*s
ImageList	... die *ImageList* mit den auf den Registerkarten anzuzeigenden Bildern
ItemSize	... bestimmt die Größe der Registerkarten
MultiLine	... legt fest, ob Karteireiter mehrzeilig sein können
Padding	... bestimmt Abstand zwischen Beschriftung und Rand eines Karteireiters
RowCount	... liest Anzahl der Zeilen eines Karteireiters
SelectedIndex	... legt Index der aktuellen Registerkarte fest
SelectedTab	... bestimmt aktuell ausgewählte Registerkarte
TabPages	... liest Auflistung der Registerkarten des *TabControl*s

BEISPIEL 32.30: *Appearance*

Appearance = Buttons



Appearance = FlatButtons



Wichtige Ereignisse

Wenn sich die aktuelle Registerkarte ändert, haben wir es mit folgender Ereigniskette zu tun:

Deselecting ⇨ *Deselected* ⇨ *Selecting* ⇨ *Selected*

Ereignisse	Beschreibung
Deselecting / Deselected	Übergabe eines Objekts vom Typ *TabControlCancelEventArgs*.
Selecting / Selected	Übergabe eines Objekts vom Typ *TabControlEventArgs*

Mit diesen Ereignissen können Sie auf einen Seitenwechsel reagieren (z.B. Sichern der Eingabewerte oder Initialisieren der enthaltenen Controls).

Mit der *Cancel*-Eigenschaft von *TabControlCancelEventArgs* können Sie den eingeleiteten Vorgang abbrechen (siehe auch *FormClosing*-Event).

Außer den oben aufgeführten Ereignissen ist noch das *SelectedIndexChanged*-Ereignis erwähnenswert, welches bei Ändern der *SelectedIndex*-Eigenschaft auftritt.

32.3.5 ImageList

Um es gleich vorwegzunehmen, das *ImageList*-Steuerelement allein ist nur von begrenztem Nutzen, es fungiert lediglich als (zur Laufzeit unsichtbarer) Container für mehrere (gleich große) Bitmaps, die Sie über ihren Index in der Liste ansprechen und in folgenden Steuerelementen anzeigen lassen können:

- *Label*, *Button*
- *ListView*, *TreeView*
- *ToolBar*
- *TabControl*

Um diese Controls mit der *ImageList* zu verbinden, setzen Sie einfach die jeweils vorhandene *ImageList*-Eigenschaft. Dies kann zur Entwurfszeit erfolgen, Sie dürfen aber auch zur Laufzeit diese Eigenschaft zuweisen.

Bevor Sie die Grafik in die Komponente einlesen, sollten Sie sich für die endgültige Größe (Eigenschaft *ImageSize*) entscheiden, da alle Grafiken auf diese Werte skaliert werden, egal wie groß sie vorher waren.

Über die *Images*-Collection rufen Sie zur Entwurfszeit einen eigenen Editor auf, in den Sie die Grafiken einfügen und, was auch wichtig ist, sie dann sortieren können.

HINWEIS: Der Umgang mit dem "Image-Auflistungs-Editor" wird u.a. im Abschnitt 32.2.20 und im Praxisbeispiel 32.10.3 beschrieben.

Möchten Sie die Grafiken zur Laufzeit laden, verwenden Sie die *Add*-Methode der *Images*-Collection.

BEISPIEL 32.31: Hinzufügen von Grafiken zur Laufzeit

C#

```
imageList1.Images.Add(new Bitmap("C:\\temp\\Bold.bmp"));
imageList1.Images.Add(new Icon("C:\\temp\\Italic.ico"));
```

BEISPIEL 32.32: Anzeige in einem Button

C#

```
button1.Text = String.Empty;
button1.ImageList = imageList1;
button1.ImageIndex = 0;
```

Ergebnis

HINWEIS: Verwenden Sie die Eigenschaft *ImageAlign*, um die Grafik in der jeweiligen Komponente auszurichten.

32.4 Menüs & Symbolleisten

In diesem Toolbox-Abschnitt finden Sie diverse komplexe Komponenten, die die bequeme Bedienbarkeit einer Anwendung ermöglichen.

32.4.1 MenuStrip und ContextMenuStrip

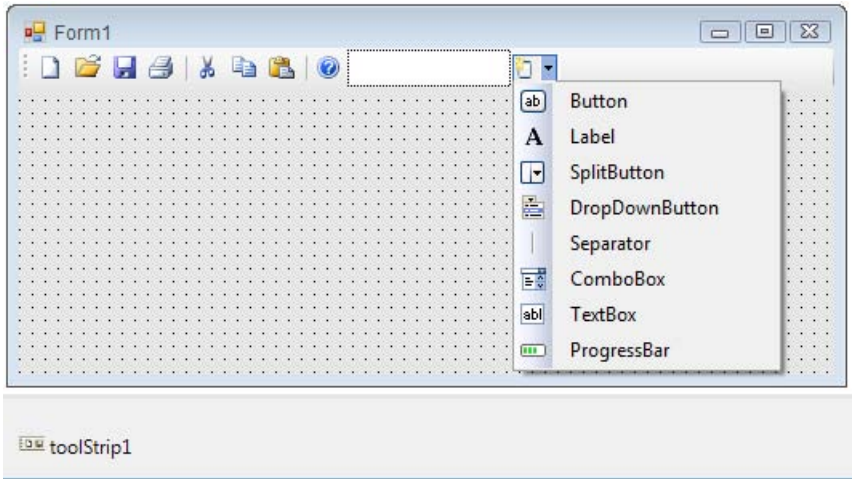
Auf beide Menü-Komponenten, die ab .NET 2.0 die veralteten *MainMenu* und *ContextMenu* abgelöst haben, wurde bereits im Vorgängerkapitel 31 ausführlicher eingegangen. Im Zusammenhang damit sind u.a. die folgenden Objekte von Bedeutung:

- *ToolStripMenuItem*
- *ToolStripComboBox*
- *ToolStripTextBox*

HINWEIS: Beachten Sie, dass sowohl *ToolStripComboBox* als auch *ToolStripTextBox* keine Untermenüs enthalten können.

32.4.2 ToolStrip

Dieses Steuerelement ist Nachfolger der *ToolBar* und erlaubt einen komfortablen Entwurf von frei konfigurierbaren Werkzeugleisten, die man mit unterschiedlichen Schalt- und Anzeigeelementen (*Button, Label, SplitButton, ProgressBar* ...) besetzen kann.



32.4.3 StatusStrip

Hier handelt es sich um den Nachfolger der altbekannten *StatusBar,* wie sie in der Regel an den unteren Rand des Formulars angedockt wird. Genauso wie das *ToolStrip* kann man diese Kompo-

nente mit unterschiedlichen Schalt- und Anzeigeelementen (*Button*, *Label*, *SplitButton*, *ProgressBar*, ...) bestücken.

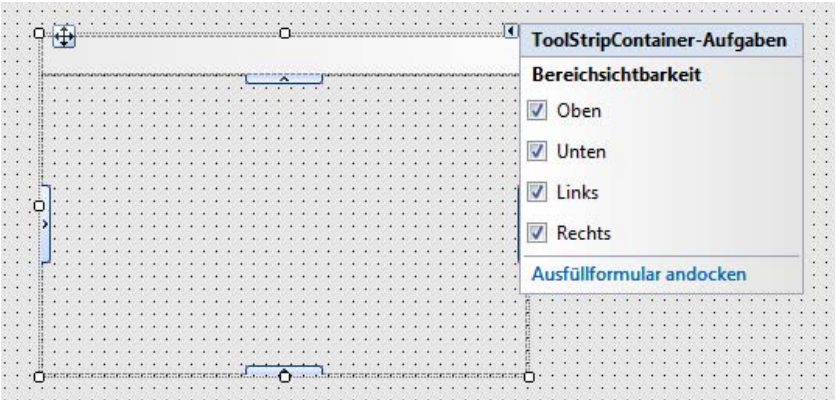
32.4.4 ToolStripContainer

Diese Container-Komponente übernimmt das flexible Positionieren von *MenuStrip* und *StatusStrip,* die zur Entwurfszeit in einen der vier Seitenbereiche (bzw. Panels) des *ToolStripContainer*s gezogen werden. Der Zentralbereich enthält die eigentlichen Steuerelemente des Formulars.

Die Seitenbereiche sind Objekte vom Typ *ToolStripPanel* und können durch Klick auf die entsprechende Markierung geöffnet werden.

In der Regel setzen Sie die *Dock*-Eigenschaft des *ToolStripContainer*s auf *Fill*, sodass diese Komponente den gesamten Clientbereich des Formulars einnimmt.

Alternativ können Sie aber auch im SmartTag-Fensterchen den Eintrag "Ausfüllformular andocken" wählen um festzulegen, an welchen Seitenrändern Menü- und Symbolleisten zur Laufzeit angedockt werden können.



32.5 Daten

Die in diesem Abschnitt enthaltenen Controls sollen in der Regel das Programmieren von ADO.NET-Anwendungen erleichtern. Ausführliche Beschreibungen und Anwendungsbeispiele finden Sie in den Kapiteln 24 und 25.

32.5.1 DataSet

Mit dieser Komponente erzeugen Sie eine Instanz eines typisierten oder aber eines nichttypisierten *DataSet*s. Das *DataSet* ist das zentrale Objekt der ADO.NET-Technologie.

32.5.2 DataGridView/DataGrid

Dieses Control hat seit ADO.NET 2.0 die Nachfolge des *DataGrid* angetreten. Letzteres steht nicht nur aus Kompatibilitätsgründen weiterhin zur Verfügung, es bietet auch den Vorteil, dass mehrere Tabellen und ihre Beziehungen gleichzeitig angezeigt werden können. Das *DataGridView* hingegen verfügt über ein deutlich umfangreicheres Objekt- und Ereignismodell, so können z.B. die Spaltenelemente auch aus anderen Komponenten bestehen, z.B. aus ComboBoxen.

HINWEIS: Das *DataGridView* ist nicht zwingend nur für den Einsatz in Datenbankanwendungen konzipiert, sondern für die tabellenförmige Darstellung nahezu beliebiger Daten geeignet.

32.5.3 BindingNavigator/BindingSource

Diese Steuerelemente sind im Zusammenhang mit der Datenbindung von Steuerelementen von Interesse und werden im Kapitel 35 besprochen.

32.5.4 Chart

Lange hat es gedauert, aber jetzt ist es endlich auch für die in die Jahre gekommenen Windows Forms soweit, Microsoft bietet eine eigene *Chart*-Komponente an, die in fast gleicher Form unter ASP.NET zur Verfügung steht.

HINWEIS: Lassen Sie sich vom ersten Eindruck der Komponente nicht täuschen, diese ist recht flexibel und kann an fast alle Wünsche des Programmierers angepasst werden.

So lassen sich mit der Komponente 35 verschiedene Diagrammtypen in 2D/3D-Darstellung anzeigen, Sie als Programmierer haben Einfluss auf Farben, Schatten, Betrachtungspunkte etc., es dürfte für jeden etwas dabei sein.

BEISPIEL 32.33: Chart mit Daten füllen

C#

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Der Standard-Reihe eine neue Bezeichnung zuweisen:

```
            chart1.Series[0].Name = "Umsätze TFT";
```

Vier Daten-Pärchen übergeben:

```
            chart1.Series[0].Points.AddXY(2007, 10);
            chart1.Series[0].Points.AddXY(2008, 25);
            chart1.Series[0].Points.AddXY(2009, 75);
            chart1.Series[0].Points.AddXY(2010, 150);
```

BEISPIEL 32.33: Chart mit Daten füllen

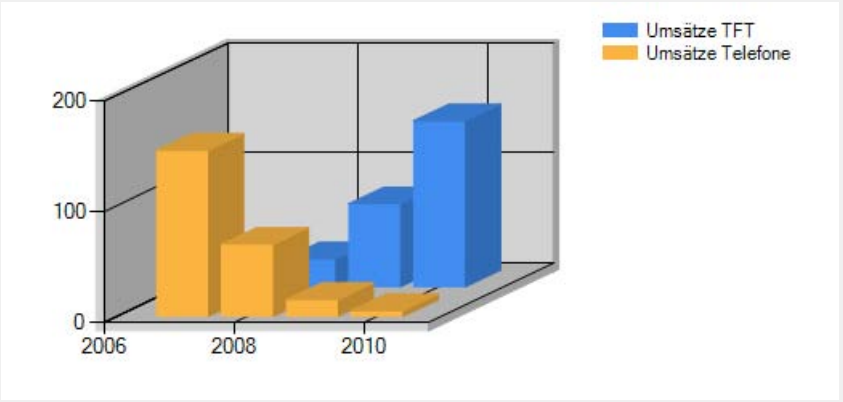
Wir erzeugen eine weitere Reihe und vergeben gleich eine Bezeichnung:

```
            chart1.Series.Add("Umsätze Telefone");
```

Auch hier wieder vier Daten-Pärchen zuweisen:

```
            chart1.Series[1].Points.AddXY(2007, 150);
            chart1.Series[1].Points.AddXY(2008, 65);
            chart1.Series[1].Points.AddXY(2009, 15);
            chart1.Series[1].Points.AddXY(2010, 5);
        }
```

Ergebnis



Wie Sie sehen, ist mit wenigen Zeilen ein recht ansprechendes Diagramm erstellt, das Sie zum Beispiel mit

```
chart1.Printing.PrintPreview();
```

in einer Druckvorschau anzeigen können oder auch gleich mit

```
chart1.Printing.Print(false);       // ohne Druckerauswahl
```

auf dem Standarddrucker ausgeben können.

Selbstverständlich ist auch eine direkte Datenbindung an diverse Datenquellen möglich, was bereits auf den Datenbankentwickler als Hauptzielgruppe hinweist.

HINWEIS: Ein umfangreiche Einführung zu dieser Komponente finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012]. Einzelne Rezepte und Lösungen können Sie auch unserem [Visual C# 2012 Kochbuch] entnehmen.

32.6 Komponenten

In diesem Abschnitt sind einige häufig benötigte Controls enthalten, die die Arbeit des Programmierers unterstützen und die nicht direkt auf dem Formular, sondern im Komponentenfach ihren Platz finden.

HINWEIS: Viele praktische Anwendungen finden Sie in unserem [Visual C# 2012 Kochbuch]!

32.6.1 ErrorProvider

Mit diesem Control können Sie auf einfache Weise eine Eingabevalidierung realisieren, um den Benutzer Ihrer Programme sofort auf ungültige Eingaben hinzuweisen (in diesem Fall erscheint ein Warnsymbol, z.B. neben einer *TextBox*).

32.6.2 HelpProvider

Hierbei handelt es sich um eine nicht sichtbare Komponente zur Einbindung der Hilfefunktionalität in ein Formular.

32.6.3 ToolTip

Über dieses Control steuern Sie, wie und vor allem wann Tooltips (die kleinen gelben Hinweisfähnchen) angezeigt werden. Der eigentliche Tooltip wird immer bei den jeweiligen Eigenschaften der Anzeige-Controls verwaltet.

32.6.4 BackgroundWorker

Diese Komponente erlaubt es Ihnen, auf einfache Weise einen Hintergrundthread zu starten (siehe Kapitel 9).

32.6.5 Timer

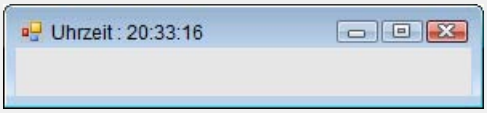
Diese Komponente löst in bestimmten Zeitabständen das *Tick*-Ereignis aus. Wesentlich ist die *Interval*-Eigenschaft, sie legt die Zeit (in Millisekunden) fest, der Wert 0 (null) ist nicht zulässig.

HINWEIS: Im Unterschied zu anderen Komponenten hat die *Enabled*-Eigenschaft standardmäßig den Wert *False*, zum Einschalten des *Timer*s müssen Sie diese Eigenschaft auf *True* setzen!

BEISPIEL 32.34: Die Uhrzeit wird per Timer im Formularkopf angezeigt (*Interval = 1000*).

C#
```
private void timer1_Tick(object sender, EventArgs e)
{
    this.Text = "Uhrzeit : " + DateTime.Now.ToLongTimeString();
}
```

Ergebnis



32.6.6 SerialPort

Dieses Control ermöglicht den Zugriff auf die serielle Schnittstelle des Rechners und ermöglicht damit die Steuerung von peripheren Geräten, die über ein solches RS 232-Interface verfügen. Mehr dazu finden Sie im Abschnitt 28.4 und in unserem [Visual C# 2012 Kochbuch].

32.7 Drucken

Relativ bescheiden ist das Angebot an Steuerelementen für alle Aktivitäten rund um die Druckausgabe (siehe dazu Kapitel 34), was jedoch nicht bedeutet, dass Sie beim Drucken nicht umfangreiche Unterstützung erfahren.

32.7.1 PrintPreviewControl

Dieses Steuerelement bündelt alle Aktivitäten rund um die Anzeige einer Druckvorschau. An dieser Stelle wollen wir allerdings nicht vorgreifen, sondern Sie gleich an das Kapitel 34 (Drucken) verweisen.

32.7.2 PrintDocument

Mit dieser Komponente sind Sie in der Lage, beliebige Grafiken zu Papier zu bringen. Den kompletten Überblick über die Verwendung finden Sie ebenfalls im Druckerkapitel 34.

32.8 Dialoge

Diese Komponenten kapseln die bekannten Windows-Dialoge.

32.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog

Hier handelt es sich um die Dialoge zum Öffnen und zum Abspeichern von Dateien, sowie zum Blättern in Verzeichnissen, mehr dazu im Datei-Kapitel (Abschnitt 7.6) bzw. im PB 8.8.1 (Zugriff auf Textdatei).

32.8.2 FontDialog/ColorDialog

Von Schriftartendialog und Farbdialog haben Sie sicher bereits während der Programmentwicklung Gebrauch gemacht, da die entsprechenden Eigenschaften bzw. Enumerationen bequem damit eingestellt werden können. Wie Sie diese Komponenten richtig einsetzen zeigt das Grafik-Kapitel 33.

32.9 WPF-Unterstützung mit dem ElementHost

Sicher hat es sich mittlerweile auch bei den eingefleischten Windows Forms-Programmierern herumgesprochen, dass sich mit WPF (*Windows Presentation Foundation*) optisch recht anspruchsvollen Anwendungen realisieren lassen (Animationen etc.), die weit über die Fähigkeiten der Windows Forms-Komponenten hinausgehen.

Möchten Sie im Rahmen Ihrer Windows Forms-Anwendung auch mit WPF-Controls arbeiten, können Sie dies über den so genannten *ElementHost* realisieren. Dazu fügen Sie Ihrem Projekt (nicht dem Form) zunächst ein WPF-Control hinzu. In dieses lassen sich beliebige WPF-Komponenten einfügen und per XAML-konfigurieren, das Control bildet lediglich den Container. Abschließend kann das neu erstellte WPF-Control einem *ElementHost* zugewiesen werden, nutzen Sie dazu die *Child*-Eigenschaft oder das Aufgabenmenü des *ElementHost*-Controls.

Das Konzept dürfte Sie an die Benutzersteuerelemente erinnern, auch hier wird zunächst in einem Container entwickelt, der abschließend in das Formular eingeblendet wird.

Ein durchgehendes Beispiel finden Sie im Praxisbeispiel

► 32.10.4 WPF-Komponenten mit dem ElementHost anzeigen

ab Seite 1609.

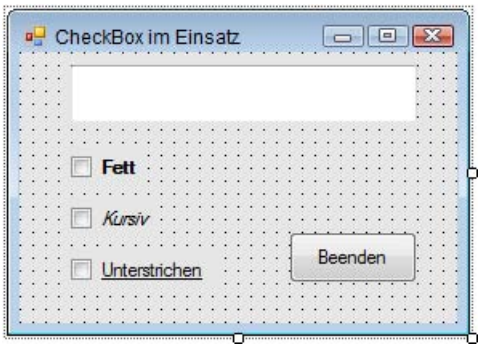
32.10 Praxisbeispiele

32.10.1 Mit der CheckBox arbeiten

Dieses kleine Beispiel demonstriert den Einsatz des *CheckBox*-Steuerelements anhand einer durchaus sinnvollen Aufgabenstellung: Der Schriftstil (fett, kursiv, unterstrichen) des Inhalts einer *TextBox* soll geändert werden. En passant wird auch der Umgang mit dem *Font*-Objekt erklärt.

Oberfläche

Nur eine *TextBox* (*MultiLine=True*), drei *CheckBox*en und ein *Button* sind erforderlich.



Quellcode

```
public partial class Form1 : Form
{ ...
```

Alle drei *CheckBox*en teilen sich einen gemeinsamen Eventhandler für das *CheckedChanged*-Ereignis. Geben Sie den Rahmencode komplett selbst ein (also nicht durch die IDE generieren lassen) und weisen Sie erst dann auf der "Ereignisse"-Seite des Eigenschaftenfensters diesen Eventhandler für jede der drei *CheckBox*en einzeln zu.

```
    private void CBChanged(object sender, EventArgs e)
    {
```

Der resultierende Schriftstil wird durch bitweise ODER-Verknüpfung schrittweise zusammengebaut:

```
        FontStyle ftStyle = FontStyle.Regular;
        if (checkBox1.Checked) ftStyle |= FontStyle.Bold;
        if (checkBox2.Checked) ftStyle |= FontStyle.Italic;
        if (checkBox3.Checked) ftStyle |= FontStyle.Underline;
```

Ein neues *Font*-Objekt wird auf Basis des geänderten Schriftstils erzeugt und der *TextBox* zugewiesen:

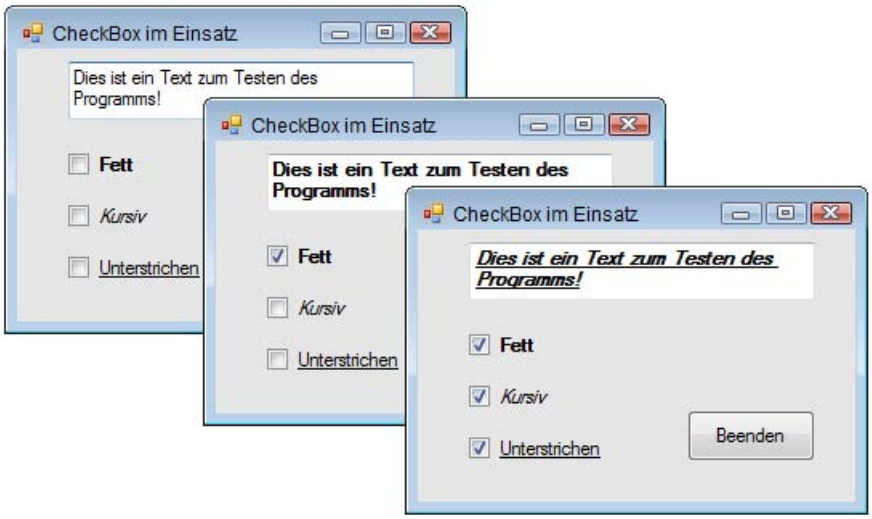
```
        textBox1.Font = new Font(textBox1.Font, ftStyle);
    }
```

Die "Beenden"-Schaltfläche:

```
    private void button1_Click(object sender, EventArgs e)
    {
        this.Close();
    }
}
```

Test

Da alle drei Fontstile miteinander kombiniert werden können, gibt es insgesamt acht Möglichkeiten (drei davon zeigt die folgende Abbildung).



32.10.2 Steuerelemente per Code selbst erzeugen

Der Windows Forms-Designer erlaubt zwar einen bequemen visuellen Entwurf von Benutzerschnittstellen, allerdings kann es sich manchmal durchaus lohnen, wenn man (zumindest teilweise) auf die Dienste des Designers verzichtet und die Steuerelemente "per Hand" programmiert. Dabei gewinnt man nicht nur einen tieferen Einblick in die Prinzipien der objekt- und ereignisorientierten Programmierung, sondern kann auch elegante Oberflächen gestalten, die Aussehen und Funktionalität zur Laufzeit ändern. Ein besonderer Vorteil ergibt sich beim Entwurf von zahlreichen, funktionell gleichartigen, Steuerelementen (Control-Arrays), denn hier bietet Ihnen der Designer kaum Unterstützung.

Die folgende Demo löst das gleiche Problem wie im Vorgängerbeispiel (Zuweisung des Schriftstils für eine *TextBox*), wobei die Steuerelemente per Code erzeugt werden. Die drei *CheckBox*en sind dabei als Steuerelemente-Array organisiert.

Oberfläche

Gönnen Sie Ihrem Windows Forms-Designer eine Pause, das nackte Formular (*Form1*) genügt!

Quellcode

```
public partial class Form1 : Form
{
```

Die benötigten globalen Variablen:

```
    private TextBox tb = new TextBox();          // Verweis auf TextBox
    private CheckBox[] checkBoxes = null;        // Verweis auf CheckBox-Array
    private int anz = 0;                         // Anzahl der CheckBoxen
```

Initialisiertes Array mit den verwendeten Schriftstilen:

```
private FontStyle[] ftStyles =
             { FontStyle.Regular, FontStyle.Bold, FontStyle.Italic, FontStyle.Underline };
```

Im Konstruktor des Formulars werden die benötigten Steuerelemente erzeugt und mit ihren wesentlichen Eigenschaften initialisiert:

```
public Form1()
{
    InitializeComponent();
```

TextBox erzeugen:

```
    int xpos = 25, ypos = 10;       // linke obere Ecke der TextBox
    tb.Location = new Point(xpos, ypos);
    tb.Multiline = true;
    tb.Width = 200; tb.Height = 40;
```

Keinesfalls vergessen darf man das Hinzufügen des Steuerelements zur *Controls*-Auflistung des Formulars (ansonsten bleibt es verborgen):

```
    this.Controls.Add(tb);
```

Alle *CheckBox*en erzeugen:

```
    anz = ftStyles.Length-1;
    checkBoxes = new CheckBox[anz];

    for (int i = 0; i < anz; i++)
    {
        checkBoxes[i] = new CheckBox();
        checkBoxes[i].Location = new Point(xpos, ypos + 50 + i * 25);
        checkBoxes[i].Text = ftStyles[i+1].ToString();
```

Gemeinsame Ereignisbehandlung zuweisen:

```
        checkBoxes[i].CheckedChanged += new EventHandler(CBChanged);
    }
```

Alle *CheckBox*en zum Formular hinzufügen

```
    this.Controls.AddRange(checkBoxes);

    tb.Text = "Dies ist ein Text zum Testen des Programms!";
}
```

Die Ereignisbehandlung für alle *CheckBox*en:

```
private void CBChanged(object sender, EventArgs e)
{
```

Zunächst normalen Schriftstil einstellen:

```
    FontStyle ftStyle = ftStyles[0];
```

Alle *CheckBox*en durchlaufen:

```
        for (int i = 0; i < anz; i++)
        {
            CheckBox cb = checkBoxes[i];
            if (cb.Checked)
```

Den resultierenden Schriftstil durch bitweises ODER zusammensetzen:

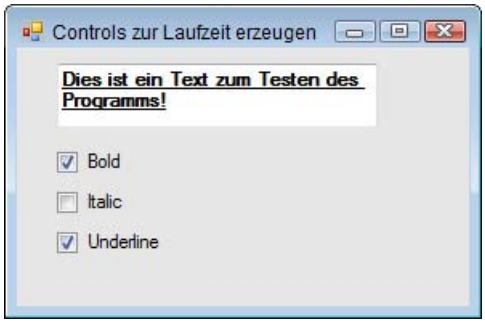
```
                ftStyle |= ftStyles[i + 1];
        }
```

Neues *Font*-Objekt erzeugen und der *TextBox* zuweisen:

```
        tb.Font = new Font(tb.Font, ftStyle);
    }
}
```

Test

Es gibt keine wesentlichen Unterschiede zum Vorgängerbeispiel. Aus Gründen der Einfachheit wurden auf eine deutsche Beschriftung der drei *CheckBox*en und auf eine "Beenden"-Schaltfläche verzichtet:



Bemerkungen

- Der Vorteil der direkten Programmierung von Steuerelemente-Arrays gegenüber deren visueller Erstellung im Designer wird umso deutlicher, je mehr Controls im Array vorhanden sind, denn der zusätzliche Code-Aufwand erhöht sich nur unwesentlich.
- Vergleichen Sie den Code mit dem des Vorgängerbeispiels und vor allem mit dem dort automatisch generierten Code von *InitializeComponent()* in *Form1.Designer.cs*.

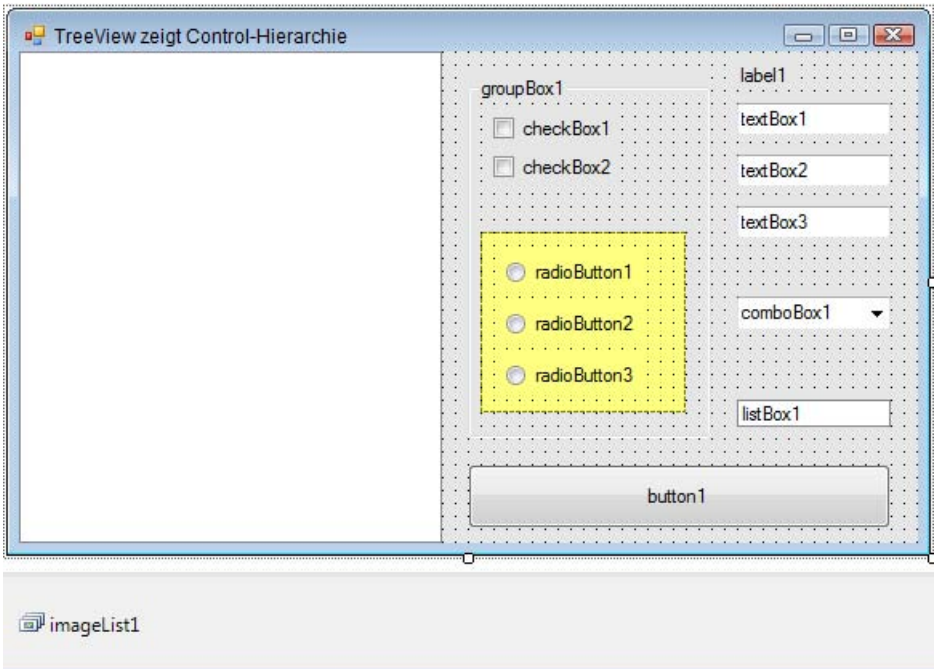
32.10.3 Controls-Auflistung im TreeView anzeigen

Das *TreeView*-Steuerelement gehört mit zu den komplexesten Windows Forms-Controls. Es ist zur Anzeige beliebig verschachtelter hierarchischer Strukturen, wie z.B. Verzeichnisbäume, bestimmt. Auch die *Controls*-Auflistung des Formulars liefert ein hervorragendes Beispiel, um den grundlegenden Umgang mit dieser Komponente zu demonstrieren.

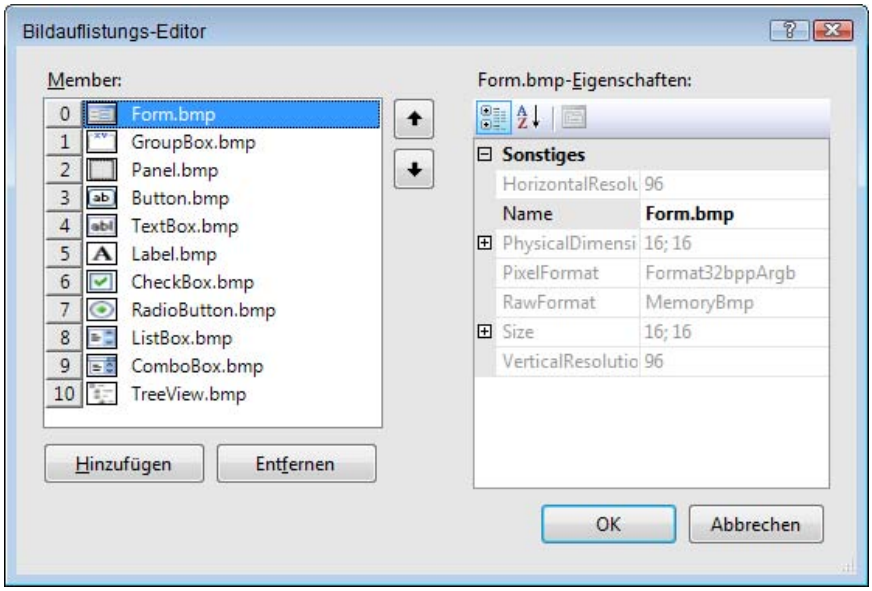
Oberfläche

Ein *TreeView*-Komponente (*Dock = Left*), eine *ListView*-Komponente sowie eine vom Prinzip her beliebige Anzahl von Steuerelementen gestalten die Oberfläche von *Form1*. Um eine aussagekräftige Tiefe der Hierarchieebenen zu erhalten, haben wir eine *GroupBox* mit aufgenommen, in welcher u.a. auch ein *Panel* enthalten ist. Beide Komponenten verfügen, genauso wie das Formular, über eigene *Controls*-Auflistungen.

Um die Strukturansicht attraktiv und übersichtlich zu gestalten, sollte jeder Knoten mit einem Icon ausgestattet werden, welches das entsprechende Steuerelement symbolisiert. Mit dem an die *ImageList* angeschlossenen "Image-Auflistungs-Editor" ist das Hinzufügen der dafür benötigten Bilddateien (16 x 16 Pixel) im Handumdrehen erledigt.



HINWEIS: Vergessen Sie nicht, die *ImageList*-Eigenschaft der *TreeView*-Komponente mit der *ImageList* zu verbinden!



Quellcode

```
public partial class Form1 : Form
{
```

Alles passiert im Konstruktor des Formulars:

```
    public Form1()
    {
        InitializeComponent();
```

TreeView mit dem Wurzelknoten initialisieren (der entspricht *Form1*):

```
        this.treeView1.Nodes.Add(this.Name);
```

Array zum Speichern der Knoten (pro Control ein Knoten):

```
        TreeNode[] nodesForm = new TreeNode[this.Controls.Count];
```

Die folgende Anweisung ist dann überflüssig, wenn sich, wie in unserem Fall, das Bildchen für *Form1* an Position 0 in der *ImageList* befindet:

```
this.treeView1.Nodes[0].ImageIndex = 0;
```

Übrige Knoten erzeugen:

```
            this.setNodes(this.Controls, this.treeView1.Nodes[0]);
    }
```

Alle Controls des Formulars werden durchlaufen um den *TreeView* aufzubauen. Um tatsächlich alle Steuerelemente zu erfassen, muss sich die *setNodes*- Methode immer wieder selbst aufrufen:

```
    private void setNodes(IList controls, TreeNode node)
    {
```

```
        foreach (Control ctrl in controls)
        {
            int index = node.Nodes.Add(new TreeNode(ctrl.Name));

            setImage(node, ctrl, index);   // das richtige Icon zuordnen

            if (ctrl.Controls.Count > 0)
            {
                setNodes(ctrl.Controls, node.Nodes[index]);    // rekursiver Aufruf!
            }
        }
    }
```

Die folgende Hilfsmethode lädt das zum Control passende Icon aus der *ImageList* und beschriftet den Knoten entsprechend:

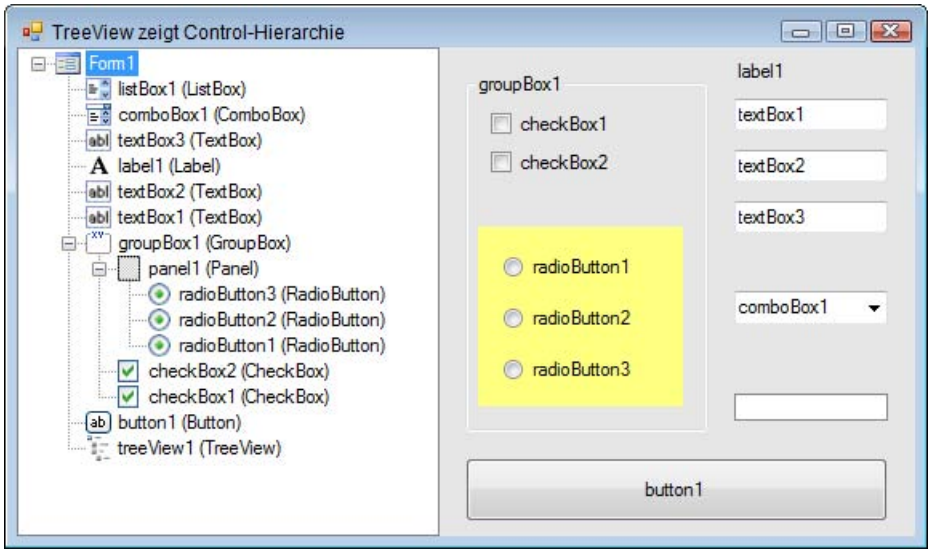
```
    private static void setImage(TreeNode node, Control ctrl, int index)
    {
        if (ctrl is GroupBox)
        {
            node.Nodes[index].ImageIndex = 1;
            node.Nodes[index].Text = ctrl.Name + " (GroupBox)";
        }
        else if (ctrl is Panel)
        {
            node.Nodes[index].ImageIndex = 2;
            node.Nodes[index].Text = ctrl.Name + " (Panel)";
        }
        else if (ctrl is Button)
        {
            node.Nodes[index].ImageIndex = 3;
            node.Nodes[index].Text = ctrl.Name + " (Button)";
        }
        else if (ctrl is TextBox)
        { ... usw.
```

Den Code für alle anderen Controls sparen wir uns hier (siehe Buchbeispiele). Es dürfte klar sein, dass der *ImageIndex* der Position entsprechen muss, auf welcher Sie das entsprechende Bildchen im "Image-Auflistungs-Editor" abgelegt haben.

```
    }
}
```

Test

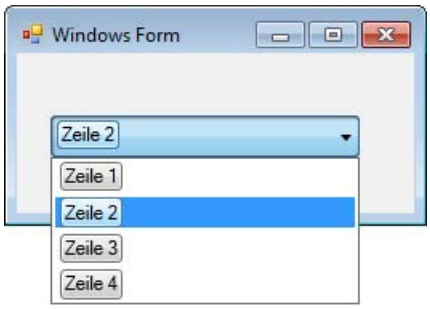
Unmittelbar nach Programmstart ist zunächst nur der Wurzelknoten (*Form1*) sichtbar. Nach Expandieren dieses Knotens erscheinen die Controls der ersten Ebene. Die zweite und dritte Ebene entstehen nach Expandieren der Knoten für *GroupBox* bzw. *Panel*.



32.10.4 WPF-Komponenten mit dem ElementHost anzeigen

Wozu WPF in einem Windows Forms-Kapitel? Die Antwort ist schnell gegeben, wenn die Migration nach WPF ansteht, oder wenn Sie Effekte realisieren wollen, die sich mit den konventionellen Windows Forms-Controls nicht umsetzen lassen.

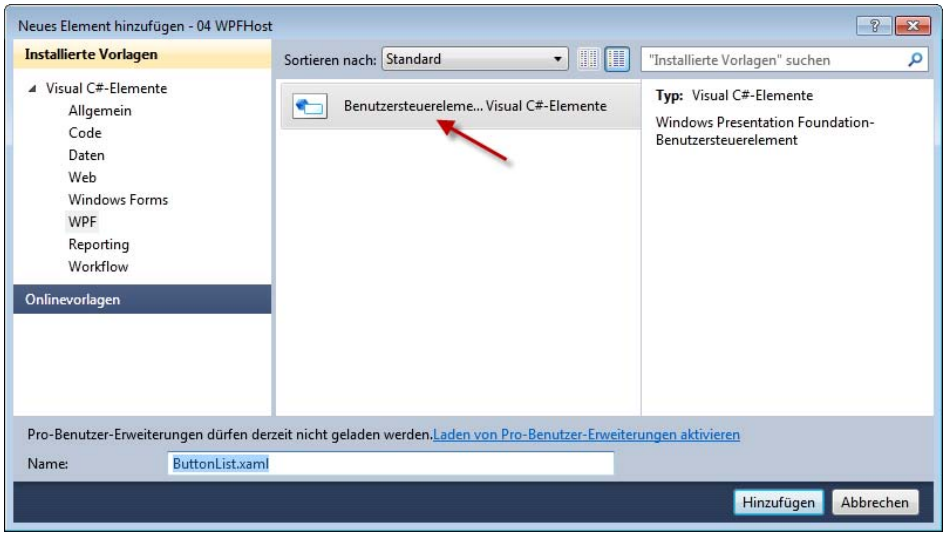
In unserem kleinen Praxisbeispiel soll eine WPF-*ComboBox* in ein Windows Forms-Formular eingefügt werden. Sicher nichts spektakuläres, aber die WPF-Controls erlauben das beliebige Verschachteln von Komponenten und so wollen wir statt einfacher *ComboBox*-Einträge vier einzelne Schaltflächen anzeigen, die auch auf einen Mausklick reagieren (siehe folgende Laufzeitansicht)[1].



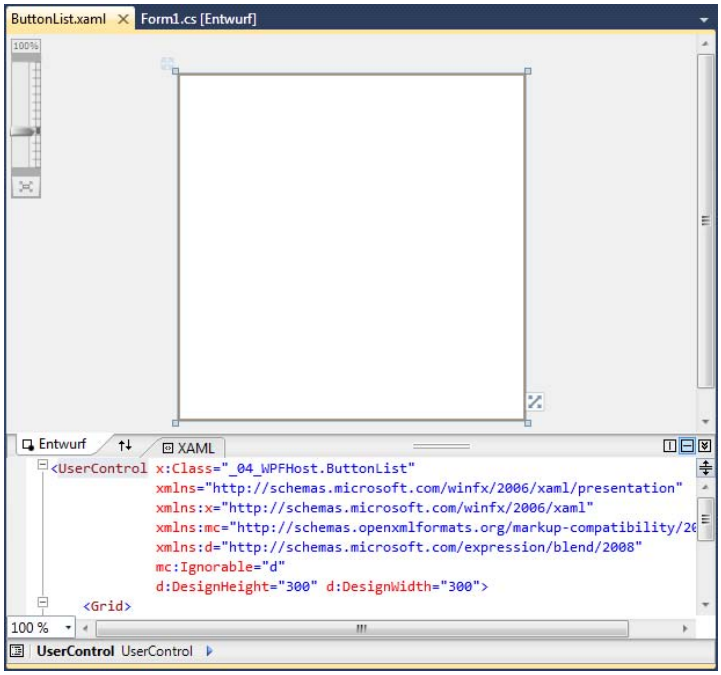
[1] Einen besonderen praktischen Nutzen hat das Beispiel sicher nicht, aber hier geht es ja um die WPF-Integration und nicht um ein unnötig kompliziertes Beispiel.

Oberfläche (WPF-Control)

In einem ersten Schritt erstellen Sie zunächst eine ganz gewöhnliche Windows Forms-Anwendung. Entgegen der sonstigen Vorgehensweise wenden wir uns noch nicht *Form1* zu, sondern fügen dem Projekt ein neues WPF-Benutzersteuerelement hinzu:



Nachfolgend öffnet sich dieses in der Entwurfsansicht:



Im oberen Teil des Designers sehen Sie den grafischen Editor, im unteren Teil den XAML-Editor, dem wir uns nachfolgend zuwenden wollen.

HINWEIS: Im Unterschied zu den Windows Forms werden die Oberflächen in WPF sprachneutral mittels XAML-Code beschrieben, weitere Details zu dieser recht umfangreichen Thematik finden Sie in den WPF-Kapiteln 12 und 13.

Erweitern Sie den bereits im Editor vorhandenen Code um die im Folgenden fett gedruckten XAML-Anweisungen:

```
<UserControl x:Class="_04_WPFHost.ButtonList"
...
    <Grid>
          <ComboBox>
              <ComboBoxItem Margin="2">
                  <Button>Zeile 1</Button>
              </ComboBoxItem>
              <ComboBoxItem Margin="2">
                  <Button Click="Button_Click">Zeile 2</Button>
              </ComboBoxItem>
              <ComboBoxItem Margin="2">
                  <Button>Zeile 3</Button>
              </ComboBoxItem>
          <ComboBoxItem Margin="2">
              <Button>Zeile 4</Button>
          </ComboBoxItem>
       </ComboBox>
   </Grid>
</UserControl>
```

An dieser Stelle nur soviel zum Code: Aufbauend auf einem Layout-Element (<*Grid*>) fügen wir eine *ComboBox* (<*ComboBox*>) mit Items (<*ComboBoxItem*>) hinzu, die wiederum Schaltflächen (<*Button*>) enthalten[1].

Beachten Sie, dass wir für den zweiten Button ein Ereignis definiert haben, das wir in der Codeansicht des WPF-Controls implementieren werden:

```
...
namespace _04_WPFHost
{
    public partial class ButtonList : UserControl
    {
...
```

Hier die hinzuzufügende Routine:

```
        private void Button_Click(object sender, RoutedEventArgs e)
```

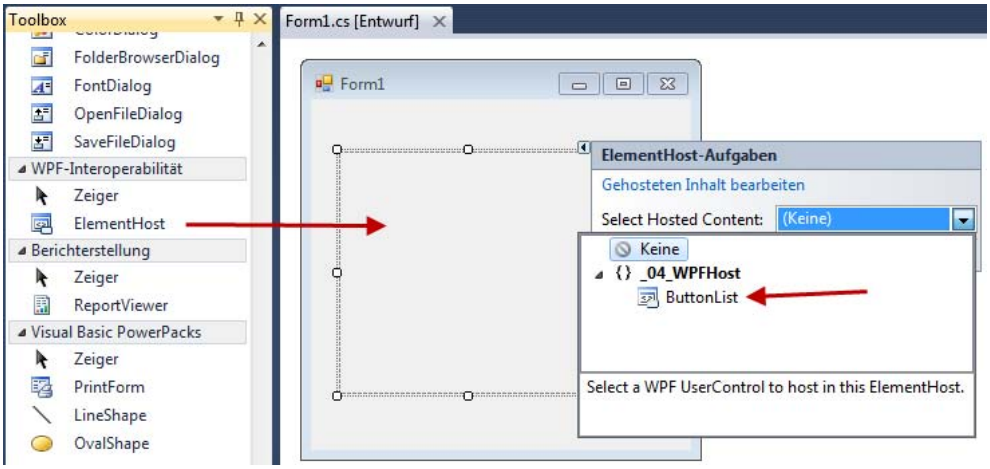
[1] Sie können diese Verschachtelung auch noch weiterführen und zum Beispiel ein Video oder ein Grid in die einzelnen Schaltflächen einfügen ...

```
        {
            MessageBox.Show("Button 2 wurde gedrückt!!!");
        }
    }
}
```

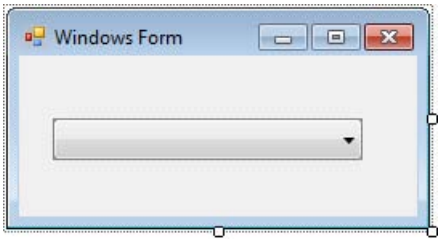
Schließen Sie nachfolgend das WPF-Benutzersteuerelement und kompilieren Sie die Anwendung, dies stellt sicher, dass Sie im nächsten Schritt auch Zugriff auf das Control haben.

Oberfläche (Windows Forms)

Öffnen Sie nun *Form1* und fügen Sie dem Formular ein *ElementHost*-Control hinzu. Über das Aufgabenmenü des Controls können Sie jetzt bequem das gewünschte WPF-Benutzersteuerelement, das wir gerade entworfen haben, hinzufügen:



Passen Sie nachfolgend noch die Größe des *ElementHost*-Controls an Ihre Wünsche an, die Größe der jetzt enthaltenen WPF-ComboBox wird dadurch direkt beeinflusst.



Damit ist der Entwurf abgeschlossen und Sie können das Programm wie gewohnt starten. Wie bereits eingangs gezeigt, können Sie zur Laufzeit die *ComboBox* aufklappen und auf die enthaltenen Schaltflächen klicken. Der Klick auf die zweite Schaltfläche löst das definierte Ereignis im WPF-Benutzersteuerelement aus.

Kapitel 33

Grundlagen Grafikausgabe

Im vorliegenden Kapitel wollen wir uns grundlegend mit dem Erzeugen, Anzeigen und Verarbeiten von Grafiken im Rahmen von Windows Forms-Anwendungen beschäftigen. C# bietet dem Programmierer drei grundsätzliche Varianten an:

- Anzeige von "vorgefertigten" Grafikdateien (z.B. Bitmaps) in der *PictureBox*-Komponente
- Einsatz von Grafikmethoden (z.B. *DrawLine*, *DrawEllipse* ...) mit dem *Graphics*-Objekt
- Arbeiten mit GDI-Funktionen

Während bei der ersten Variante die Grafik bereits zur Entwurfszeit (Design Time) entsteht, wird sie bei den beiden letzteren erst zur Laufzeit (Run Time) erzeugt. Erst hier kann man von eigentlicher "Grafikprogrammierung" sprechen, da diese ausschließlich per Quellcode (also ohne Zuhilfenahme der visuellen Entwicklungsumgebung von C#) funktioniert.

HINWEIS: Auch wenn in diesem Kapitel bereits die Grundlagen für die Druckausgabe gelegt werden, die eigentliche Beschreibung der Vorgehensweise und der nötigen Komponenten finden Sie erst im Kapitel 34.

Wem die Ausführungen dieses Kapitels nicht weit genug gehen, oder wer bereits die Grundlagen der Grafikprogrammierung beherrscht, den verweisen wir gleich an das Kapitel 36 weiter. Dort gehen wir gezielt auf einige Spezialthemen rund um die Grafikprogrammierung ein.

33.1 Übersicht und erste Schritte

Nachdem mit dem GDI (*Graphics Design Interface*) die in Win32-Anwendungen übliche Grafikausgabe etwas in die Jahre gekommen war, wurde mit der Einführung des .NET-Frameworks auch eine neue Grafikschnittstelle unter dem Namen GDI+ implementiert.

HINWEIS: Mittlerweile ist auch GDI+ "veraltet" und hat mit WPF einen neuen Nachfolger gefunden. Die Verwendung von WPF erfordert allerdings einen kompletten Umstieg auf WPF-Anwendungen, mehr dazu ab Kapitel 12).

33.1.1 GDI+ – Ein erster Einblick für Umsteiger

GDI+ erfordert schon auf Grund seiner objektorientierte Schnittstelle ein ganz anderes Vorgehen, als bei den alten GDI-Anwendungen. Umsteiger von "alten" Programmiersprachen haben es also erfahrungsgemäß etwas schwerer, sich mit den Konzepten von GDI+ anzufreunden. Aus diesem Grund zunächst einige Anmerkungen zu den Grundkonzepten von GDI+ für den Umsteiger.

Ein zentrales Grafikausgabe-Objekt

Im Gegensatz zur Vorgehensweise in GDI, bei der Sie die Grafikmethoden von einzelnen Komponenten genutzt haben, arbeiten Sie jetzt mit einem einzigen *Graphics*-Objekt, das Sie bestimmten Objekten zuordnen können bzw. von diesen ableiten. Nur dieses Objekt verfügt über relevante Grafikmethoden und -eigenschaften.

Die Grafikausgabe ist zustandslos

Eine der wichtigsten und zugleich einschneidendsten Änderungen betrifft die Organisation der Grafikausgabe. Haben Sie bisher zunächst die Parameter für Linien (Breite, Farbe, Style etc.), Pinsel oder Schriftarten gesetzt und nachfolgend mit diesen Grafikausgaben getätigt, ist dafür nun jede einzelne Grafikmethode zuständig. D h., für eine Grafikmethode ist es vollkommen unerheblich, welche Linienart vorher genutzt wurde, alle erforderlichen Parameter werden an die Methode bei **jedem** Aufruf übergeben. Die gleiche Vorgehensweise trifft auch auf Texte oder Pinsel zu.

Prinzipieller Ablauf

Wurde bisher nach dem Schema

- Grafikobjekte (Pen, Brush etc.) erzeugen,
- beliebige Zeichenfunktionen (z.B. LineTo) aufrufen,
- Grafikobjekte löschen

verfahren, müssen bei einer zustandslosen GDI+-Programmierung genau diese Operationen **für jeden** einzelnen Methodenaufruf ausgeführt werden, da GDI+ die GDI-Objekte im alten Zustand zurücklassen muss.

BEISPIEL 33.1: Zeichnen einer Linie im Formular

C#

```
Graphics g;
```

Neuen Stift erzeugen:

```
Pen mypen = new Pen(Color.Aqua);
```

Graphics-Objekt für das Formular erzeugen:

```
g = this.CreateGraphics();
```

Linie mit dem neuen Stift zeichnen:

```
g.DrawLine(mypen, 10, 10, 100, 100);
```

Wichtige Features

Die wichtigsten Features auf einen Blick:

- GDI+ bietet eine verbesserte Farbverwaltung sowie mehr vordefinierte Farben.
- GDI+ unterstützt eine breite Palette an Bildformaten (.bmp, .gif, .jpeg, .exif, .png, .tiff, .ico, .wmf, .emf).
- Antialiasing-Funktionalität
- Farbverläufe für Pinsel
- Splines
- Bildtransformationen (Rotation, Translation, Skalierung)
- Gleitkommaunterstützung
- Unterstützung des Alpha-Kanals und damit auch Alpha-Blending
- ...

HINWEIS: Dass mit WPF der Nachfolger bereits in den Startschuhen steht haben wir bereits erwähnt. Dies sollte jedoch nicht darüber hinweg täuschen, dass Windows Forms-Anwendungen und damit GDI+ auch die nächsten Jahre noch aktuell bleiben, zumal es für diesen Bereich auch schon reichlich fertige Lösungen gibt.

33.1.2 Namespaces für die Grafikausgabe

Bevor Sie im weiteren Verlauf mit der Fehlermeldung "Der Typ XYZ ist nicht definiert." konfrontiert werden, möchten wir Ihnen die im Zusammenhang mit der Grafikausgabe relevanten Namespaces vorstellen:

- *System.Drawing*
- *System.Drawing.Drawing2D*
- *System.Drawing.Imaging*
- *System.Drawing.Printing*
- *System.Drawing.Design*
- *System.Drawing.Text*

Die ausführliche Behandlung der einzelnen Strukturen und Objekte erfolgt in den weiteren Abschnitten, hier nur eine kurze Übersicht.

System.Drawing

Dieser Standardmäßig eingebundene Namespace enthält die meisten Klassen, Typen, Auflistungen etc., die Sie für die Basisfunktionen der Grafikausgabe benötigen.

Typ	Beschreibung
Color	... verwaltet ARGB-Farbwerte (Alpha-Rot-Grün-Blau), Konvertierungsfunktionen sowie diverse vordefinierte Farbkonstanten.
Point *PointF*	... verwaltet 2-D-Koordinaten (x, y) als Integer- bzw. als Floatwert.
Rectangle *RectangleF*	... verwaltet die Koordinaten eines Rechtecks als Integer- bzw. Floatwert.
Size	... verwaltet die Größe eines rechteckigen Bereichs (Breite, Höhe).

Objekt	Beschreibung
Graphics	**Das zentrale Grafikausgabe-Objekt**
Pen	Objekt für die Definition des Linientyps
Brush, *Brushes,* *SolidBrush,* *TextureBrush*	Objekte für die Definition des Füllstils (Pinsel) von Objekten (Kreise, Rechtecke etc.)
Font, *FontFamily*	Objekt für die Definition von Schriftarten (Farbe, Größe etc.)
Bitmap, Image	Objekt für die Verwaltung von Bitmaps bzw. Grafiken

System.Drawing.Drawing2D

Dieser Namespace bietet Funktionen für Farbverläufe sowie Unterstützung für 2-D- und Vektorgrafiken (Matrix für geometrische Transformationen).

System.Drawing.Imaging

Mit Hilfe dieses Namespace können Sie erweiterte Funktionen wie

- Direktzugriff auf Bitmap-Daten (Pointer),
- Unterstützung für Metafiles,
- Farbverwaltung,
- Grafikkonvertierung,
- Abfrage von Grafikeigenschaften

realisieren.

System.Drawing.Printing

Basisklassen für die Druckausgabe. Mehr dazu im Kapitel 34 (Druckausgabe).

System.Drawing.Design

Dieser Namespace enthält einige Klassen, die für die Entwurfszeit-Oberfläche genutzt werden.

System.Drawing.Text

Abfrage von Informationen über die installierten Schriftarten.

33.2 Darstellen von Grafiken

Bevor wir Sie in den folgenden Abschnitten mit Dutzenden von Objekten, Methoden und Eigenschaften überfrachten, wollen wir uns die wohl trivialste Form der Grafikdarstellung etwas näher ansehen. Die Rede ist von der *PictureBox*-Komponente, die in diesem Zusammenhang alle wesentlichen Aufgaben übernehmen kann. Untrennbar mit dieser Komponente ist auch das *Image*-Objekt verbunden, über das die eigentlichen Grafikdaten verwaltet werden (Laden, Manipulieren, Eigenschaften).

33.2.1 Die PictureBox-Komponente

Platzieren Sie zunächst eine *PictureBox* im Formular, können Sie diese wie jedes andere Objekt positionieren und die Größe bestimmen. Über die Eigenschaft *Image* lässt sich bereits zur Entwurfszeit eine Grafikdatei zuweisen, die nachfolgend fest in das Projekt und damit auch die EXE-Datei übernommen wird.

Wie diese Grafik formatiert, d h. skaliert wird, bestimmen Sie mit der *SizeMode*-Eigenschaft:

Normal

CenterImage

StretchImage

Autosize

Wie Sie den obigen Abbildungen entnehmen können, skaliert die *Stretch*-Variante zwar die Grafik auf die gewünschte Größe, leider bleibt dabei aber das Seitenverhältnis der Grafik auf der Strecke.

Wie Sie die Proportion zur Laufzeit wiederherstellen können, zeigt das folgende Beispiel.

BEISPIEL 33.2: Bild proportional skalieren

C#

```
Single xy;
xy = (Single) pictureBox1.Image.Width / (Single) pictureBox1.Image.Height;
if (pictureBox1.Image.Width > pictureBox1.Image.Height)
  pictureBox1.Height = (int) (pictureBox1.Width / xy);
else
  pictureBox1.Width = (int) (pictureBox1.Height * xy);
```

Die Vorgehensweise: Sie weisen der Eigenschaft *SizeMode* den Wert *Stretch* zu und skalieren jeweils die Außenmaße der *PictureBox* so, dass die Grafik möglichst optimal angezeigt wird.

Über die *BorderStyle*-Eigenschaft können Sie zwischen *keinem*, *einfachem* und *dreidimensionalem* Rahmen wählen.

Damit sind auch schon alle wesentlichen Eigenschaften der *PictureBox* beschrieben. Ziemlich dürftig, werden Sie sicher sagen. Aber dieser Eindruck täuscht, wenn wir uns die bereits erwähnte Eigenschaft *Image* näher ansehen, bei der es sich um ein recht komplexes Objekt handelt.

33.2.2 Das Image-Objekt

Mit dem *Image*-Objekt bietet sich nicht nur die Möglichkeit, zur Entwurfszeit eine Grafik in die *PictureBox* einzubetten, sondern es stellt auch diverse Grafikbearbeitungsfunktionen und die dazu nötigen Informationen über die Grafik bereit.

Zwei Wege führen zum *Image*-Objekt:

- Haben Sie bereits zur Entwurfszeit eine Grafik geladen, können Sie direkt über *Picture.Image* auf die gewünschten Eigenschaften/Methoden zugreifen.
- Andernfalls müssen Sie zunächst ein *Image*-Objekt erzeugen und der *PictureBox* zuweisen.

BEISPIEL 33.3: Erzeugen eines neuen Image-Objekts (eine Bitmap 100 x 100 Pixel)[1]

C#

```
Image img;
img = new Bitmap(100, 100);
pictureBox1.Image = img;
```

Alternativ lässt sich ein *Image* auch aus einer Grafikdatei erzeugen:

```
Image img;
img = Image.FromFile("c:\\test.jpg");
pictureBox1.Image = img;
```

[1] Dies ist nur eine Variante!

Womit wir auch schon beim Laden von Grafiken angekommen sind.

33.2.3 Laden von Grafiken zur Laufzeit

Wie bereits im vorhergehenden kurzen Beispiel gezeigt, stellt es kein Problem dar, ein *Image* aus einer bereits existierenden Datei zu laden. Je nach Dateityp handelt es sich beim *Image* nachfolgend um eine Pixel-, Vektorgrafik (WMF, EMF) oder ein Icon (ICO).

HINWEIS: Doch Vorsicht: Nicht jedes Grafikformat unterstützt auch alle Grafikmethoden des *Image*-Objekts. Laden Sie beispielsweise eine WMF-Grafik in das Image, können Sie diese nicht mit der Funktion *RotateFlip* drehen oder spiegeln.

BEISPIEL 33.4: Grafikformat bestimmen

C#

Statt mit

```
Image img;
img = Image.FromFile("c:\\test.bmp");
pictureBox1.Image = img;
```

... können Sie auch direkt den gewünschten *Image*-Typ erzeugen:

```
Image img;
img = Bitmap.FromFile("c:\\test.bmp");
pictureBox1.Image = img;
```

Das Resultat ist in beiden Fällen das Gleiche.

33.2.4 Sichern von Grafiken

Ähnlich einfach wie das Laden ist auch das Speichern von Grafiken. Mit der *Image*-Methode *Save* können Sie die Grafik in einem der unterstützten Dateiformate sichern.

BEISPIEL 33.5: Speichern im PNG-Format

C#

```
pictureBox1.Image.Save("c:\\test.png");
```

Auf diese Weise können Sie auch einen Dateikonverter programmieren, Sie brauchen nicht einmal eine *PictureBox* dafür.

BEISPIEL 33.6: Konvertieren vom BMP- ins PNG-Format

C#

```
using System.Drawing.Imaging;
...
      Image img;
      img = Bitmap.FromFile("c:\\test.bmp");
      img.Save("c:\\test.png", ImageFormat.Png);
```

Spezielle Einstellungen

Leider, und das scheint bei fast allen universellen Bibliotheken der Fall zu sein, ist die praktische Verwendung teilweise recht eingeschränkt bzw. umständlich. Möchten Sie beispielsweise im JPEG-Format speichern, ist dies kein Problem, doch was, wenn Sie auch den Kompressionsfaktor beeinflussen wollen?

In diesem Fall kommen Sie um etwas mehr Programmierung nicht herum.

BEISPIEL 33.7: Festlegen eines Kompressionsfaktors von 60% für die zu speichernde JPEG-Datei

C#

```
private void button1_Click(object sender, EventArgs e)
{
  EncoderParameters myEncoderParameters = new EncoderParameters(1);
  myEncoderParameters.Param[0] = new EncoderParameter(Encoder.Quality, (long) 60);
  pictureBox1.Image.Save("c:\\test.jpg", GetEncoderInfo("image/jpeg"), myEncoderParameters);
  pictureBox2.Image = Bitmap.FromFile("c:\\test.jpg");
}
```

Die notwendige Hilfsfunktion für die Rückgabe des passenden *ImageCodecInfo*-Objekts:

```
private ImageCodecInfo GetEncoderInfo(String mt)
{
  ImageCodecInfo[] encoders = ImageCodecInfo.GetImageEncoders();
  for (int i=0; i < encoders.Length - 1; i++)
  if (encoders[i].MimeType == mt) return encoders[i];
  return null;
}
```

33.2.5 Grafikeigenschaften ermitteln

Breite und Höhe der Grafik

Breite und Höhe der Grafik können Sie über die Eigenschaften *Width* und *Height* des ***Image***-Objekts abrufen.

HINWEIS: Die Maße des *Image*-Objekts, d h. der Grafik, stehen in keinem Zusammenhang mit den Maßen der *PictureBox*.

Auflösung

Die vertikale bzw. horizontale Auflösung der Grafik können Sie mit den Eigenschaften *VerticalResolution* bzw. *HorizontalResolution* abfragen. Beide geben einen Wert in *dpi* (Punkte pro Inch) zurück.

Grafiktyp

Den aktuell geladenen Grafiktyp fragen Sie über die Eigenschaft *Rawformat* ab. Rückgabewerte sind die verschiedenen Bildtypen, die von der *Image*-Komponente unterstützt werden (BMP, GIF etc.).

Interner Bitmap-Aufbau

Um welchen Typ von Bitmap (Farbtiefe) bzw. um welches Speicherformat (RGB, ARGB etc.) es sich handelt, verrät Ihnen die Eigenschaft *PixelFormat*. An dieser Stelle handelt es sich jedoch lediglich um eine schreibgeschützte Eigenschaft. Möchten Sie Einfluss auf das Speicherformat nehmen, sollten Sie sich näher mit dem Kapitel 36 beschäftigen.

33.2.6 Erzeugen von Vorschaugrafiken (Thumbnails)

Sicher sind Ihnen unter Windows auch schon die kleinen Vorschaugrafiken in den Explorerfenstern aufgefallen. Bevor Sie jetzt an aufwändige Algorithmen und Funktionen zur Skalierung von Bitmaps denken, vergessen Sie es besser wieder. GDI+ unterstützt Sie an dieser Stelle mit einer einfach verwendbaren Methode:

```
SYNTAX: Image GetThumbnailImage(int thumbWidth,int thumbHeight,
                                Image.GetThumbnailImageAbort callback, IntPtr callbackData);
```

Die beiden ersten Parameter dürften leicht verständlich sein, *callback* erwartet einen Verweis auf einen Delegate (optional *null). CallbackData* übergeben Sie grundsätzlich *IntPtr.Zero*.

Leider müssen Sie sich um die Proportionen des zu erzeugenden Image selbst kümmern. Übergeben Sie beispielsweise 100 für Breite und Höhe und haben Sie eine Ausgangsgrafik mit anderem Höhen-/Seitenverhältnis, wird die Vorschaugrafik gestaucht, was sicher nicht sehr professionell aussieht. Fragen Sie also vorher das Höhen-/Seitenverhältnis der Grafik ab und setzen Sie Breite und Höhe entsprechend.

BEISPIEL 33.8: Funktion zum Erzeugen einer proportionalen Vorschaugrafik

C#

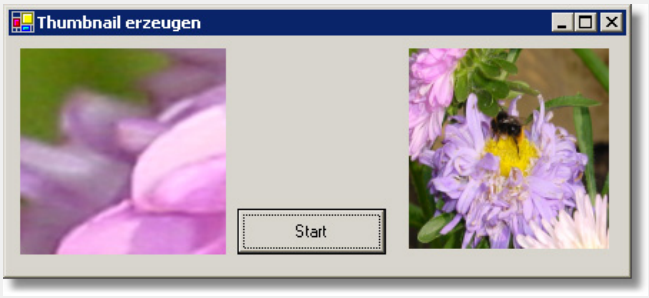
```
private Image GetThumb(int size,Image img)
{
  Single xy = (Single) (img.Width / img.Height);
  if (xy > 1)
     return img.GetThumbnailImage(size, (int) (size / xy), null, IntPtr.Zero);
  else
     return img.GetThumbnailImage((int)(size * xy), size, null, IntPtr.Zero);
}
```

Der Aufruf ist einfach:

```
private void button1_Click(object sender, EventArgs e)
{
   pictureBox2.Image = GetThumb(80, pictureBox1.Image);
}
```

BEISPIEL 33.8: Funktion zum Erzeugen einer proportionalen Vorschaugrafik

Ergebnis



HINWEIS: Enthält die Ausgangsgrafik bereits eine Thumbnail-Grafik (z.B. bei JPG-Format), wird diese für die Methode *GetThumbnailImage* genutzt, um nicht die ganze Bitmap in den Speicher zu laden. Dies ist allerdings maximal bis zu einer Auflösung von 120 x 120 Pixeln sinnvoll. Benötigen Sie größere Thumbnails, sollten Sie besser die Methode *DrawImage* verwenden, um eine verkleinerte Kopie der Ausgangsbitmap zu erzeugen.

33.2.7 Die Methode RotateFlip

Mit der Methode *RotateFlip* stehen Ihnen rudimentäre Manipulationsfunktionen für die Grafikausgabe zur Verfügung (drehen, spiegeln). Übergeben Sie der Methode einfach die gewünschte Konstante und aktualisieren Sie gegebenenfalls die übergeordnete *PictureBox*-Komponenten, schon ist die Grafik gedreht oder gespiegelt.

BEISPIEL 33.9: Drehen um 180°, vertikal kippen

C#

```
pictureBox1.Image.RotateFlip(RotateFlipType.Rotate180FlipY);
pictureBox1.Refresh();
```

Ein "Schweizer Taschenmesser", was die Funktionalität anbelangt, könnte man auf den ersten Blick denken, doch wer sich die Liste der verschiedenen Möglichkeiten (Konstanten) einmal näher ansieht wird feststellen, dass einige Varianten schlicht überflüssig sind. Ausgangspunkt für die Beispielgrafiken ist folgende "aufwändige" Grafik:

Konstante	Beschreibung	Resultat
Rotate180FlipNone	Drehung um 180 Grad	
Rotate180FlipX	Drehung um 180 Grad, horizontal kippen	
Rotate180FlipXY	Drehung um 180 Grad, horizontal und vertikal kippen	

Konstante	Beschreibung	Resultat
Rotate180FlipY	Drehung um 180 Grad, vertikales kippen	
Rotate270FlipNone	Drehung um -90 Grad	
Rotate270FlipX	Drehung um -90 Grad, horizontal kippen	
Rotate270FlipXY	Drehung um -90 Grad, horizontal und vertikal kippen	
Rotate270FlipY	Drehung um -90 Grad, vertikal kippen	
Rotate90FlipNone	Drehung um 90 Grad	
Rotate90FlipX	Drehung um 90 Grad, horizontal kippen	
Rotate90FlipXY	Drehung um 90 Grad, horizontal und vertikal kippen	
Rotate90FlipY	Drehung um 90 Grad, vertikal kippen	
RotateNoneFlipX	horizontal kippen	
RotateNoneFlipXY	horizontal und vertikal kippen	
RotateNoneFlipY	vertikal kippen	

Eine Rotation um 180° (*Rotate180FlipNone*) entspricht einem Kippen in vertikaler und horizontaler Richtung (*RotateNoneFlipXY*). Auch bei der Kombination *Rotate90FlipXY* bzw. *Rotate270-FlipXY* scheint mit den Entwicklern der Spieltrieb durchgegangen zu sein, das gleiche Ergebnis kann auch mit *Rotate270FlipNone* bzw. mit *Rotate90FlipNone* erreicht werden.

Unangefochtener "Spitzenreiter" in der Tabelle ist *Rotate180FlipXY.*

33.2.8 Skalieren von Grafiken

Eine Grafik mit GDI+ zu skalieren stellt eines der kleinsten Probleme dar[1]. Mit einer einzigen Zeile Code können Sie beispielsweise die Größe einer Grafik verdreifachen.

BEISPIEL 33.10: Bitmap auf dreifache Größe skalieren

C#

```
pictureBox1.Image = new Bitmap(pictureBox1.Image,
                    new Size(pictureBox1.Image.Width * 3, pictureBox1.Image.Height * 3));
```

Wir nehmen diese Anweisung jetzt noch einmal auseinander bzw. deklarieren die Objekte für eine bessere Übersicht:

[1] Sieht man einmal von potenziellen Speicherproblemen bei großen Bitmaps ab.

BEISPIEL 33.10: Bitmap auf dreifache Größe skalieren

Eine temporäre Bitmap deklarieren:

```
Bitmap b;
```

Eine Variable für die neue Größe deklarieren:

```
Size s;
```

Die neue Größe bestimmen:

```
s.Width = pictureBox1.Image.Width * 3;
s.Height = pictureBox1.Image.Height * 3;
```

Die neue Bitmap erzeugen:

```
b = new Bitmap(PictureBox1.Image, s);
```

Die Bitmap dem Image zuweisen:

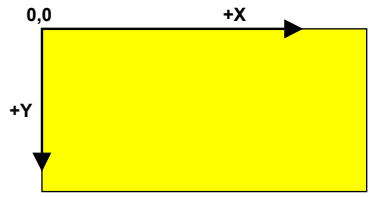
```
pictureBox1.Image = b;
```

Damit sind die wichtigsten Möglichkeiten von *PictureBox* und *Image* aufgezählt, wir können uns der eigentlichen Grafikprogrammierung, d h. dem Erzeugen von Grafiken, zuwenden.

33.3 Das .NET-Koordinatensystem

Bevor Sie sich mit dem Zeichnen von Linien, Kreisen oder der Ausgabe von Bitmaps beschäftigen, sollten Sie einen Blick auf das .NET-Koordinatensystem werfen.

Auf den ersten Blick bietet sich nichts Ungewohntes, der Koordinatenursprung für alle Grafikmethoden liegt in der linken oberen Ecke des jeweils gewählten Ausgabefensters (das kann auch ein Steuerelement sein). Positive x-Werte werden nach rechts, positive y-Werte nach unten abgetragen:



Die Maßeinheit für Ausgaben ist standardmäßig das Pixel. Alternativ kann es jedoch bei einer Druckausgabe von Vorteil sein, die Maßeinheit zum Beispiel in Millimeter zu ändern. Nutzen Sie dazu die Eigenschaft *PageUnit* des *Graphics*-Objekts. Folgende Werte sind zulässig:

Wert	Beschreibung
Display	1/100 Zoll
Document	1/300 Zoll

Wert	Beschreibung
Inch	Zoll (2,54 cm)
Millimeter	Millimeter
Pixel	(Standard) Bildschirmpixel
Point	1/72 Zoll

GDI+ unterscheidet im Zusammenhang mit der Grafikausgabe drei verschiedene Koordinatensysteme:

- globale Koordinaten,
- Seitenkoordinaten und
- Gerätekoordinaten.

33.3.1 Globale Koordinaten

Hierbei handelt es sich um das ursprüngliche Koordinatensystem. Dieses wird über bestimmte Transformationen in Seiten- und damit auch Gerätekoordinaten umgewandelt.

Ausgangspunkt bei der Maßeinheit Pixel: Ein Pixel in globalen Koordinaten entspricht zunächst einem Pixel auf dem endgültigen Ausgabegerät (Bildschirm, Drucker).

BEISPIEL 33.11: Globale Koordinaten

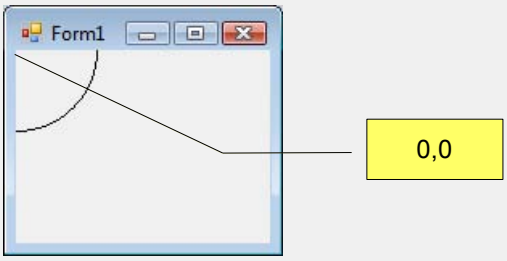
C#

Es soll ein Kreis mit dem Radius 50 (Pixel) auf dem Formular gezeichnet werden (an dieser Stelle müssen wir leider etwas vorgreifen, mit den folgenden Anweisungen wird im *Paint*-Ereignis des Formulars der Kreis gezeichnet):

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis auf dem Bildschirm:



33.3.2 Seitenkoordinaten (globale Transformation)

Möchten Sie den Nullpunkt des Koordinatensystems verschieben, sodass der gesamte Kreis sichtbar wird, müssen Sie den Nullpunkt verschieben, d.h., eine globale Transformation durchführen.

Translation (Verschiebung)

Mit Hilfe der Methode *TranslateTransform* lässt sich diese Aufgabe bewältigen.

BEISPIEL 33.12: Translation

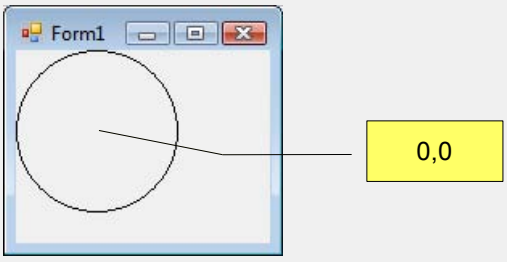
C#

Verschieben des Nullpunkts um jeweils 50 Pixel in x- und y-Richtung

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   e.Graphics.TranslateTransform(50, 50);
   e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

Ergebnis

Das Ergebnis der Grafikausgabe:



Skalierung (Vergrößerung/Verkleinerung)

Ähnlich verhält es sich mit einer Skalierung zwischen globalen und Seitenkoordinaten: Möchten Sie beispielsweise den Kreis auf dem Bildschirm doppelt so groß ausgeben, können Sie dies mit der Eigenschaft *PageScale* steuern.

BEISPIEL 33.13: Skalierung

C#

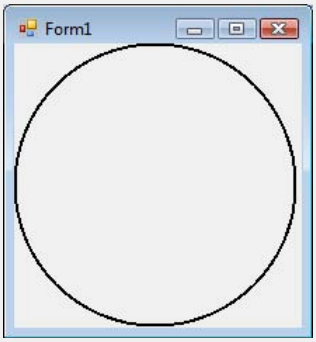
Ein Pixel global soll zwei Pixeln Seiteneinheit entsprechen

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    e.Graphics.TranslateTransform(50, 50);
    e.Graphics.PageScale = 2;
    e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
}
```

BEISPIEL 33.13: Skalierung

Ergebnis

Wie Sie sehen, sind alle Zeichenanweisungen, inklusive der Strichstärke, von dieser Skalierung betroffen:



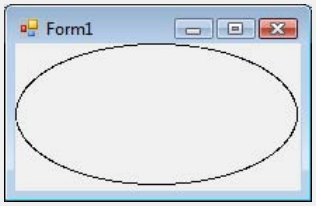
Alternativ können Sie mit *ScaleTransform* auch unterschiedliche Skalierungsfaktoren für x und y einführen.

BEISPIEL 33.14: Skalierung in x-Richtung (zweifach)

C#

```
e.Graphics.ScaleTransform(2, 1);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

Ergebnis



HINWEIS: Sollen auch Images im korrekten Verhältnis wiedergegeben werden, müssen Sie *ScaleTransform* verwenden!

Rotation

Als letzte Variante bleibt das Drehen des Koordinatensystems. Verantwortlich dafür ist die Methode *RotateTransform*, der Sie einfach den Drehwinkel übergeben.

BEISPIEL 33.15: Drehen des Koordinatensystems um 45° (Uhrzeigersinn)

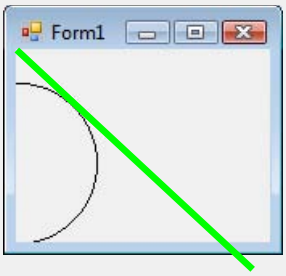
C#

```
e.Graphics.RotateTransform(45);
e.Graphics.TranslateTransform(50, 50);
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

BEISPIEL 33.15: Drehen des Koordinatensystems um 45° (Uhrzeigersinn)

Ergebnis

Die folgende Abbildung zeigt das Resultat, die eingezeichnete Linie deutet die Position der x-Achse (y = 0) an:



Dass nicht nur einfache Objekte wie Linien oder Kreise von den Transformationen betroffen sind, zeigt das folgende Beispiel.

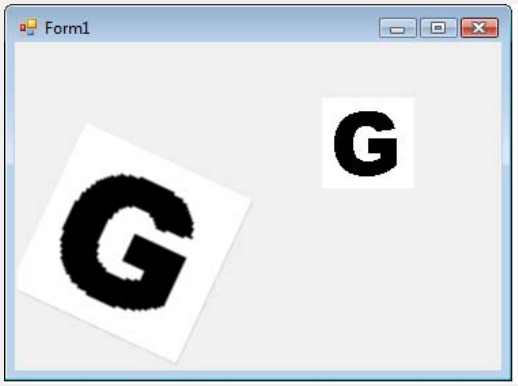
BEISPIEL 33.16: Bitmap-Transformation

C#

Drehen einer Bitmap um 25°, skalieren um 100%, verschieben des Nullpunkts in das Formular (50, 50)

```
e.Graphics.TranslateTransform(50, 50);
e.Graphics.RotateTransform(25);
e.Graphics.ScaleTransform(2, 2);
e.Graphics.DrawImage(pictureBox1.Image, 0, 0);
```

Ergebnis



33.3.3 Gerätekoordinaten (Seitentransformation)

Das dritte Koordinatensystem bezieht sich auf das endgültige Ausgabegerät, z.B. ein Drucker oder ein Bildschirm. Diese Geräte verfügen über vorgegebene Auflösungen, die in Dots per Inch (dpi) angegeben werden. Beispielsweise verfügt der Bildschirm meist über eine Auflösung von 96 dpi, Laserdrucker werden mit Auflösungen von 300 bis weit über 1000 dpi angeboten.

Solange Sie nicht mit Geräteeinheiten arbeiten, werden Sie je nach Geräteauflösung zu unterschiedlichen Ergebnissen kommen. Aus diesem Grund unterstützt das .NET-Koordinatensystem auch gerätespezifische Einheiten wie Millimeter oder Inch. Über die Eigenschaft *PageUnit* können Sie die Einheit für Ihr Koordinatensystem ändern.

BEISPIEL 33.17: Bildschirmanzeige in Millimetern

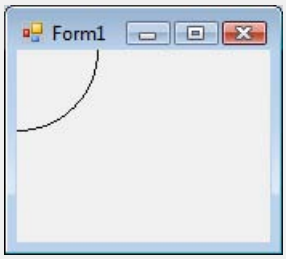
C#

```
e.Graphics.PageUnit = GraphicsUnit.Millimeter;
e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100);
```

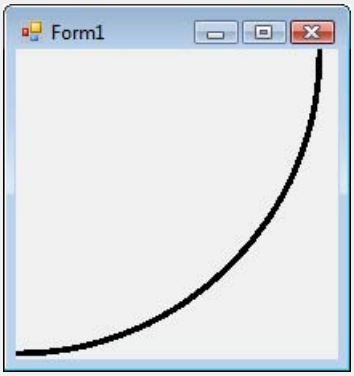
Ergebnis

Die beiden folgenden Abbildungen zeigen das Ergebnis für Pixel bzw. Millimeter:

Pixel



Millimeter



Frage: Wie groß ist bei der rechten Abbildung der Radius in Pixeln?

Die Antwort: Bei einer Bildschirmauflösung von 96 dpi ergibt sich ein Faktor von ca. 3,7 Pixeln pro Millimeter. Multipliziert mit dem Radius von 50 mm erhalten Sie 188 Pixel. Sie können das berechnete Ergebnis zum Beispiel mit Paintbrush "nachmessen".

HINWEIS: Weitere Anwendungsbeispiele für Koordinatentransformationen finden Sie in den Kapiteln 34 und 36.

33.4 Grundlegende Zeichenfunktionen von GDI+

Im vorhergehenden Abschnitt hatten Sie ja bereits ersten Kontakt mit einigen Zeichenfunktionen, auf die wir nun in diesem Abschnitt näher eingehen wollen.

33.4.1 Das zentrale Graphics-Objekt

Wie bereits angedeutet, sind mit GDI+ die Zeiten des unbekümmerten Programmierens vorbei. Um den ausufernden Möglichkeiten etwas Einhalt zu gebieten und eine einheitliche Schnittstelle für die Grafikausgabe zu schaffen, wurde die *Graphics*-Klasse eingeführt.

Nur Objekte dieses Typs verfügen über Grafikausgabemethoden und die nötigen Parameter. Alle anderen Komponenten, und dazu zählen auch Formulare und Drucker, können lediglich *Graphics*-Objekte zur Verfügung stellen.

Wie erzeuge ich ein Graphics-Objekt?

Drei wesentliche Varianten bieten sich an:

- Für Formulare können Sie das ***Paint*-Ereignis** nutzen, der übergebene Parameter *e* stellt auch ein *Graphics*-Objekt zur Verfügung.
- Für alle anderen Komponenten nutzen Sie die ***CreateGraphics*-Methode** des jeweiligen Objekts. Diese gibt ein *Graphics*-Objekt zurück.
- Möchten Sie auch auf *Images* (BMP etc.) per *Graphics*-Objekt zugreifen, verwenden Sie einfach die ***Graphics.FromImage*-Methode**.

BEISPIEL 33.18: Zeichnen mit *Paint*-Ereignis

C#
```
private void Form1_Paint(object sender,PaintEventArgs e)
{
  e.Graphics.DrawEllipse(new Pen(Color.Black), -50, -50, 100, 100) ;
}
```

BEISPIEL 33.19: Verwenden von *CreateGraphics*

C#
```
private void button1_Click(object sender, EventArgs e)
{
  Graphics g;
  g = this.CreateGraphics();
  g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
}
```

Um die Ausgabe auf einem Button statt auf dem Formular vorzunehmen, genügt es, wenn Sie das *Graphics*-Objekt mit *button1.CreateGraphics* erzeugen.

BEISPIEL 33.20: Verwenden von *Graphics.FromImage*

C#
```
private void button1_Click(object sender, EventArgs e)
{
   Graphics g;
   g = Graphics.FromImage(pictureBox1.Image);
   g.DrawEllipse(new Pen(Color.Black), 0, 0, 100, 100);
   pictureBox1.Refresh();
}
```

HINWEIS: Für alle Grafikausgaben gilt: Nach dem Verdecken eines Formulars müssen Sie sich um das Aktualisieren der Bildschirmanzeige selbst kümmern. Einzige Ausnahme ist die *PictureBox*, wenn Sie, wie im letzten Beispiel gezeigt, über *Graphics* in das *Image* schreiben.

Die Invalidate-Methode

Nutzen Sie das *Paint*-Ereignis für die Ausgabe der Grafik und muss diese zwischenzeitlich aktualisiert werden, können Sie die Methode *Invalidate* zum Aktualisieren der Anzeige verwenden.

BEISPIEL 33.21: Mittels *Timer* soll ein kontinuierlich größer werdender Kreis gezeichnet werden

C#

Globale Variable für den Offset:

```
private int x;
```

Das *Tick*-Ereignis:

```
private void timer1_Tick(object sender, EventArgs e)
{
 x++; this.Invalidate();
}
```

Die eigentliche Zeichenroutine im *Paint*-Ereignis:

```
private void Form1_Paint(object sender,PaintEventArgs e)
{
  e.Graphics.DrawEllipse(new Pen(Color.Black), 0, 0, 100 + x, 100 + x);
}
```

Im Weiteren wollen wir uns mit den wichtigsten Grafikgrundoperationen beschäftigen, auch wenn Sie bereits einige in den vorhergehenden Beispielen kennen gelernt haben.

Die Eigenschaft ResizeRedraw

So bequem die Verwendung des *Paint*-Ereignisses auch ist, einen Nachteil werden Sie schnell bemerken, wenn das Formular skaliert bzw. kurzzeitig verdeckt wurde. In diesem Fall werden nur die neuen bzw. die verdeckten Flächen neu gezeichnet. Dies kann allerdings dazu führen, dass Ihre Grafik nicht mehr wie gewünscht ausgegeben wird. Um diesem Missstand abzuhelfen, steht über das Formular die Eigenschaft *ResizeRedraw* zur Verfügung. Ist diese auf *true* gesetzt, wird immer der komplette Clientbereich neu gezeichnet.

BEISPIEL 33.22: Auswirkung von *ResizeRedraw*

C#

Zeichnen einer Ellipse im Clientbereich des Formulars:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   g.DrawEllipse(new Pen(Color.Black),0, 0, this.ClientSize.Width,this.ClientSize.Height);
}
```

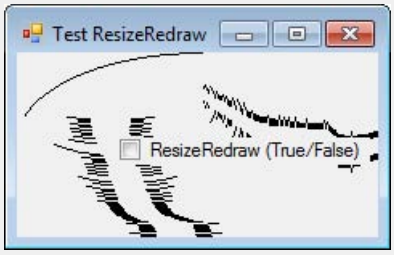
BEISPIEL 33.22: Auswirkung von *ResizeRedraw*

Per *CheckBox* die *ResizeRedraw-Eigenschaft* beeinflussen:

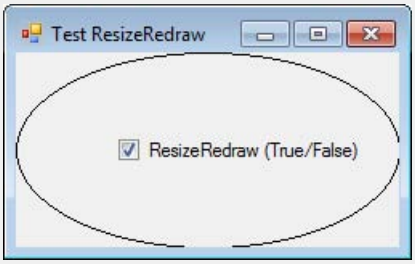
```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
    this.ResizeRedraw = checkBox1.Checked;
}
```

Ergebnis

ResizeRedraw=false:



ResizeRedraw=true:



33.4.2 Punkte zeichnen/abfragen

Tja, da sieht es gleich ganz düster aus. Eine entsprechende Funktion existiert zunächst nicht. Einzige Ausnahme: die *GetPixel-*/*SetPixel*-Methoden des *Bitmap*-Objekts.

SYNTAX: `Color GetPixel(int x, int y)`

SYNTAX: `void SetPixel(int x, int y, Color color)`

Beiden Methoden übergeben Sie die gewünschten Koordinaten, *GetPixel* gibt Ihnen einen *Color*-Wert zurück, *SetPixel* müssen Sie einen *Color*-Wert übergeben.

HINWEIS: Mehr zu Farben und dem *Color*-Objekt finden Sie in Abschnitt 33.5.3.

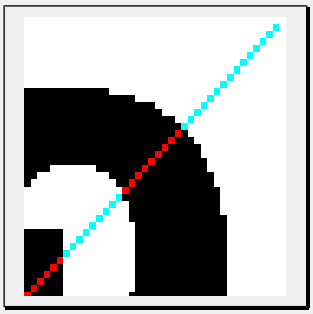
BEISPIEL 33.23: Setzen von Pixeln in Abhängigkeit von der vorhergehenden Farbe

C#

```
private void button1_Click(object sender, EventArgs e)
    {
      Bitmap b = (Bitmap) pictureBox1.Image;
      for (int x = 1; x <=50; x++)
      {
        if (b.GetPixel(x, x).ToArgb() == Color.Black.ToArgb())
          b.SetPixel(x, x, Color.Red);
        else
          b.SetPixel(x, x, Color.Aqua);
      }
      pictureBox1.Refresh();
    }
```

Ergebnis

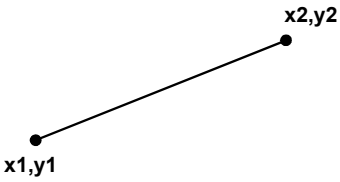
Die folgende Ausschnittsvergrößerung zeigt das Resultat:



33.4.3 Linien

Das Zeichnen von Linien gestaltet sich mit GDI+ relativ einfach, sieht man einmal davon ab, dass Sie bei **jedem** Aufruf von *DrawLine* auch einen geeigneten *Pen* übergeben müssen. An dieser Stelle wollen wir zunächst noch nicht auf die verschiedenen Stifttypen eingehen, mit denen Sie sowohl Farbe, Muster, Endpunkte etc. beeinflussen können. Mehr Infos zu diesem Thema finden Sie erst in Abschnitt 33.5.

Die weiteren Parameter der *DrawLine*-Methode: Entweder Sie übergeben jeweils ein Paar x,y-Koordinaten oder Sie verwenden gleich die vordefinierte Struktur *Point* (eine x- und eine y-Koordinate).



BEISPIEL 33.24: Verwendung von *DrawLine*

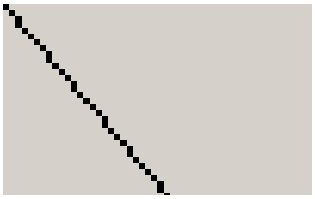
C#

```
Graphics g = this.CreateGraphics();
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
```

Mit *New Pen(Color.Black)* erzeugen wir einen schwarzen Stift mit der Linienbreite 1.

33.4.4 Kantenglättung mit Antialiasing

Im Zusammenhang mit der Ausgabe von Linien kommen wir auch schnell mit einem ersten "Problem" in Berührung. Die Rede ist von der "Treppchen"- bzw. "Stufen"-Bildung von Linien, die einen von 0° bzw. 90° verschiedenen Winkel aufweisen:



Ursache ist die begrenzte Bildschirmauflösung, es können nur die Pixel gesetzt werden, die in das xy-Raster passen. Zwangsläufig kommt es bei einer Abweichung von Raster und gewünschtem Linienverlauf zu Sprüngen, die sich als hässliche Treppchen bemerkbar machen.

Mit Hilfe der Antialiasing-Technik können diese Effekte vermindert werden. Hintergrund dieser Technik ist ein "Glätten" des Linienrandes durch Auffüllen mit Pixeln, deren Farbe aus Linien- **und** Hintergrundfarbe berechnet wird.

GDI+ bietet Ihnen zu diesem Zweck die Methode *SetSmoothingMode*, mit der Sie die Form der Kantenglättung beeinflussen können. Zwei Extreme sind möglich:

- *SmoothingModeHighSpeed* (schnell, aber kantig)
- *SmoothingModeHighQuality* (langsam(er), aber sauber)

BEISPIEL 33.25: Ein- und Ausschalten der Kantenglättung

C#

```
Graphics g = this.CreateGraphics();
g.SetSmoothingMode(SmoothingModeHighQuality);
g.DrawLine(new Pen(Color.Black), 10, 10, 100, 100);
g.SetSmoothingMode(SmoothingModeHighSpeed);
```

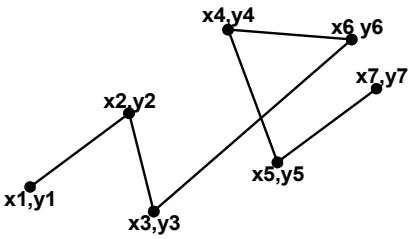
Ergebnis

Die folgende Abbildung zeigt das Ergebnis:

HINWEIS: Diese Form der Kantenglättung wirkt sich nicht auf Textausgaben, sondern nur auf alle Arten von Linien aus. Für Textausgaben verwenden Sie die Eigenschaft *TextRenderingHint*.

33.4.5 PolyLine

Möchten Sie mehr als eine Linie zeichnen und stimmen die Endpunkte mit den Anfangspunkten der folgenden Linien überein, verwenden Sie zum Zeichnen von Linien am besten die Methode *DrawLines*. Übergabewert ist wie gewohnt ein initialisierter *Pen* sowie ein Array von Punkten (*Point* oder *PointF*).



BEISPIEL 33.26: Zeichnen von zwei Linien

C#

```
Graphics g = this.CreateGraphics();
PointF[] punkte = {new PointF(0, 0), new PointF(100, 100), new PointF(20, 80)};
g.DrawLines(new Pen(Color.Black), punkte);
```

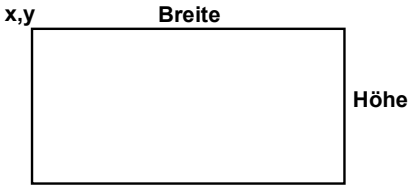
HINWEIS: Selbstverständlich können Sie auch ein frei definiertes Array verwenden, das zur Laufzeit mit Werten gefüllt wird.

33.4.6 Rechtecke

Für das Zeichnen von Rechtecken können Sie entweder die Methode *DrawRectangle* (nur Rahmen) oder *FillRectangle* (gefülltes Rechteck) verwenden.

DrawRectangle

Diese Methode erwartet als Übergabeparameter einen *Pen* sowie die Koordinaten (als x, y, Breite, Höhe oder *Rectangle*-Struktur).



BEISPIEL 33.27: Zeichnen von Rechtecken

C#
```
Graphics g = this.CreateGraphics();
g.DrawRectangle(new Pen(Color.Black), 10, 10, 100, 100);
```

Alternativ können Sie auch die folgenden Anweisungen verwenden:

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);
g.DrawRectangle(new Pen(Color.Black), rec);
```

FillRectangle

Im Unterschied zur *DrawRectangle*-Methode erwartet *FillRectangle* einen initialisierten *Brush*, d.h. die Information, wie das Rechteck zu füllen ist.

HINWEIS: Die Methode füllt lediglich das Rechteck, es wird kein Rahmen gezeichnet!

BEISPIEL 33.28: Rotes Rechteck zeichnen

C#
```
Graphics g = this.CreateGraphics();
g.FillRectangle(new SolidBrush(Color.Red), 10, 10, 100, 100);
```

Wie auch bei *DrawRectangle* können Sie alternativ eine *Rectangle*-Struktur übergeben. Dies bietet sich beispielsweise an, wenn man das Rechteck auch mit einem Rahmen versehen will.

BEISPIEL 33.29: Verwendung *Rectangle*

C#
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 100, 100);

g.FillRectangle(new SolidBrush(Color.Red), rec);
g.DrawRectangle(new Pen(Color.Black), rec);
```

DrawRectangles/FillRectangles

Nicht genug der Pein, neben den beiden genannten Methoden können Sie auch *DrawRectangle**s*** und *FillRectangle**s*** verwenden, um gleich mehrere Rechtecke auf einmal zu zeichnen. Übergeben wird in diesem Fall ein Array von *Rectangle*-Strukturen.

BEISPIEL 33.30: Zeichnen zweier gefüllter Rechtecke

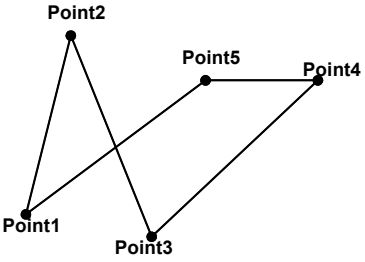
C#
```
Graphics g = this.CreateGraphics();
Rectangle[] rec = {new Rectangle(10, 10, 100, 100),
                   new Rectangle(50, 50, 200, 120)};

g.FillRectangles(new SolidBrush(Color.Red), rec);
```

BEISPIEL 33.30: Zeichnen zweier gefüllter Rechtecke

Ergebnis

33.4.7 Polygone

Möchten Sie ein n-Eck zeichnen, nutzen Sie die Methode *DrawPolygon*. Soll dieses Vieleck auch gefüllt werden, können Sie dies mit *FillPolygon* realisieren. Beide Funktionen arbeiten ähnlich wie die Funktionen zum Zeichnen von Rechtecken, der einzige Unterschied: *DrawPolygon* bzw. *FillPolygon* erwarten als Übergabeparameter ein Array von *Point*-Strukturen (x- und y-Koordinate) sowie jeweils einen *Pen* bzw. einen *Brush*.

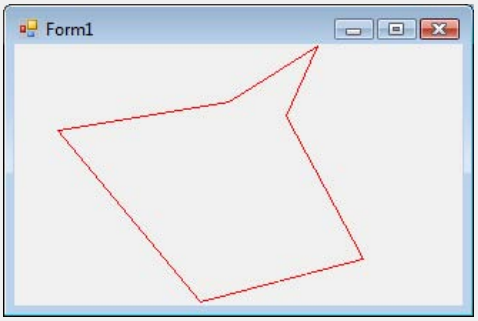


BEISPIEL 33.31: Zeichnen eines Vielecks

C#

```
Graphics g = this.CreateGraphics();
Point[] Ps = new Point[6];
Ps[0] = new Point(30, 60);   Ps[1] = new Point(150, 40);
Ps[2] = new Point(212, 1);   Ps[3] = new Point(190, 50);
Ps[4] = new Point(244, 150); Ps[5] = new Point(130, 180);
g.DrawPolygon(new Pen(Color.Red), Ps);
```

Ergebnis



HINWEIS: Die letzte Linie, zurück zum ersten Punkt, wird automatisch mit gezeichnet, um die Figur zu schließen.

33.4.8 Splines

Möchten Sie Diagramme zeichnen, stehen Sie häufig vor dem Problem, dass die entstehende Kurve ziemlich eckig ist, da Sie aus Zeitgründen nur wenige Stützpunkte berechnet haben. Das lineare Verbinden ist in diesem Fall nicht der ideale Weg. Besser ist hier die Verwendung von Splines, d h. Linienzügen, die bei einem Wechsel des Anstiegs weiche Übergänge realisieren.

GDI+ bietet Ihnen die Methode *DrawCurve*, der Sie zunächst die gleichen Parameter wie *DrawLines* übergeben können (Stift, Point-Array).

SYNTAX:
```
void DrawCurve(Pen pen, Point[] points, int Start, int Anzahl, float Spannung);
```

Zusätzlich können Sie bestimmen, ab welchem Punkt in der Liste die Spline-Kurve gezeichnet wird bzw. wie viele Punkte gezeichnet werden. Der letzte Parameter bestimmt die Spannung, d.h., wie der Übergang zwischen zwei Teilstrecken hergestellt wird. Ein Beispiel soll für mehr Klarheit sorgen.

BEISPIEL 33.32: Spline zeichnen

C#

Neues *Graphics*-Objekt erzeugen:

```
Graphics g = this.CreateGraphics();
```

Array für die Punkte erstellen:

```
PointF[] Ps = new PointF[6];
```

Die Spannung, verändern Sie diesen Wert wie gewünscht:

```
Single u = 0.6F;

Ps[0] = new PointF(0, 100);
Ps[1] = new PointF(100, 0);
Ps[2] = new PointF(200, 100);
Ps[3] = new PointF(300, 200);
Ps[4] = new PointF(400, 100);
Ps[5] = new PointF(500, 0);
```

Zunächst zeichnen wir zum Vergleich eine "normale" Kurve:

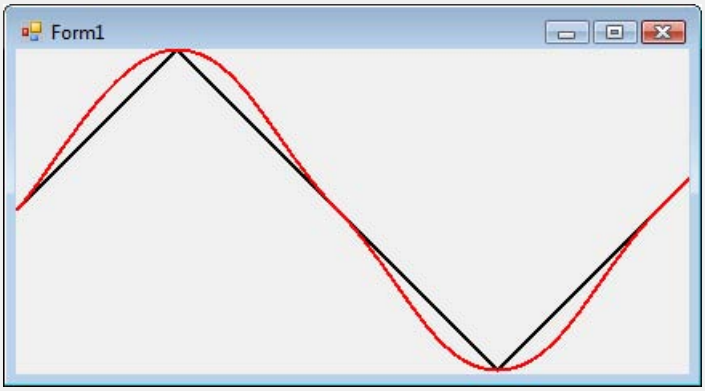
```
g.DrawLines(new Pen(Color.Black, 2), Ps);
```

Und jetzt die Spline-Kurve:

```
g.DrawCurve(new Pen(Color.Red, 2), Ps, 0, 5, u);
```

BEISPIEL 33.32: Spline zeichnen

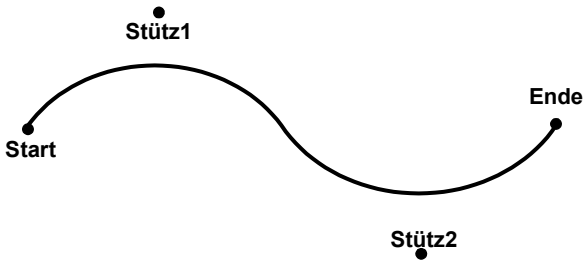
Ergebnis



33.4.9 Bézierkurven

Für einfache Zeichnungen dürften die bisherigen Zeichenfunktionen ausreichen, kompliziertere Gebilde bekommen Sie mit der Bézierfunktion in den Griff. Wie bei einer "normalen" Linie übergeben Sie Anfangs- und Endpunkt, zusätzlich jedoch noch zwei Stützpunkte, mit denen die Linie wie ein Gummiband "gezogen" werden kann.

SYNTAX: `void DrawBezier(Pen pen, Point Start, Point Stütz1, Point Stütz2, Point Ende);`



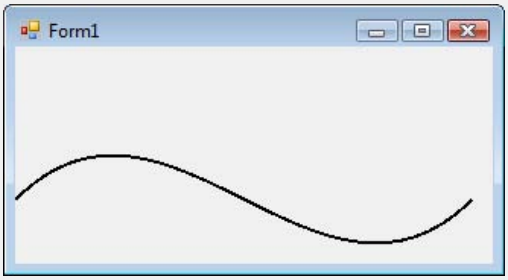
BEISPIEL 33.33: Zeichnen einer Bézierkurve

C#

```
Graphics g = this.CreateGraphics();
Point start, Stütz1, stütz2, ende;

start = new Point(0, 100);
Stütz1 = new Point(100, 0);
stütz2 = new Point(200, 200);
ende = new Point(300, 100);
g.DrawBezier(new Pen(Color.Black, 2), start, Stütz1, stütz2, ende);
```

BEISPIEL 33.33: Zeichnen einer Bézierkurve

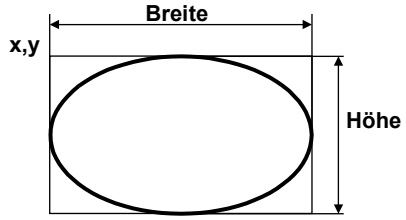
Ergebnis



HINWEIS: Mit *DrawBeziers* können Sie mehrere Bézierkurven gleichzeitig zeichnen.

33.4.10 Kreise und Ellipsen

Kreise und Ellipsen zeichnen Sie mit *DrawEllipse* bzw. *FillEllipse*, wenn Sie eine Kreis-/Ellipsenfüllung erzeugen wollen. Übergabeparameter ist eine *Rectangle*-Struktur bzw. die linke obere Ecke sowie die Breite und Höhe der Ellipse.



HINWEIS: Kreisbögen und Kuchenstücke zeichnen Sie mit eigenen Methoden.

BEISPIEL 33.34: Zeichnen einer Ellipse

C#

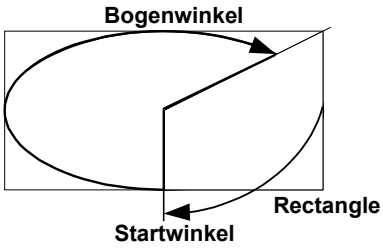
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawEllipse(new Pen(Color.Black, 2), rec);
```

33.4.11 Tortenstück (Segment)

Die Methode *DrawPie* zeichnet ein Tortenstück, das durch eine Ellipse und zwei Linien begrenzt ist. Die Begrenzungslinien werden über den Startwinkel bzw. den Bogenwinkel bestimmt.

SYNTAX: `void DrawPie(Pen pen, RectangleF rect, float startwinkel, float bogenwinkel);`

HINWEIS: Winkel werden im Uhrzeigersinn abgetragen.



HINWEIS: Mit *FillPie* erstellen Sie ein gefülltes Tortenstück.

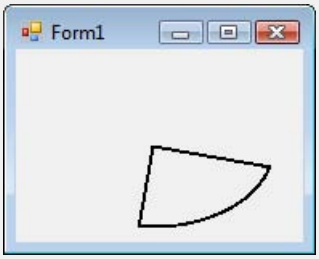
BEISPIEL 33.35: Tortenstück zeichnen

C#

```
Graphics g = this.CreateGraphics();

Rectangle rec = new Rectangle(10, 10, 150, 100);
g.DrawPie(new Pen(Color.Black, 2), rec, 10, 90);
```

Ergebnis

Zeichnet folgendes Tortenstück:



Kuchendiagramme

Da die Funktionen Winkel als Parameter erwarten, gestaltet sich das Zeichnen von Kuchendiagrammen besonders leicht. Ausgehend von einer einheitlichen *Rectangle*-Struktur brauchen Sie lediglich die Winkelangaben zu variieren.

BEISPIEL 33.36: Zeichnen eines Kuchendiagramms (100°, 60°, 200°)

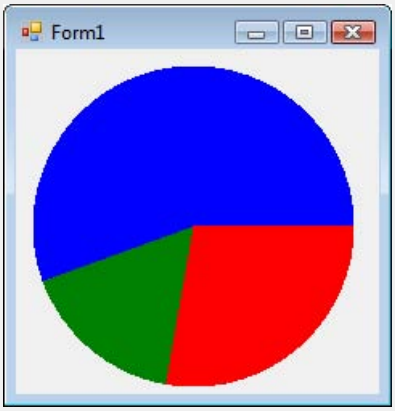
C#

```
Graphics g = this.CreateGraphics();

Rectangle rec = new Rectangle(10, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
g.FillPie(new SolidBrush(Color.Green), rec, 100, 60);
g.FillPie(new SolidBrush(Color.Blue), rec, 160, 200);
```

BEISPIEL 33.36: Zeichnen eines Kuchendiagramms (100°, 60°, 200°)

Ergebnis

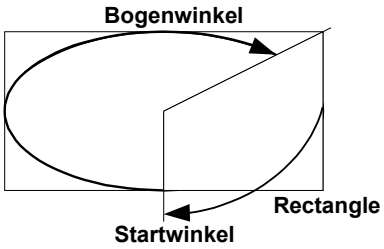
Das Ergebnis kann sich durchaus sehen lassen:



33.4.12 Bogenstück

Die Methode *DrawArc* zeichnet im Gegensatz zur Methode *DrawPie* nur den Bogen, nicht die Verbindungen zum Ellipsenmittelpunkt. Aus diesem Grund kann diese Figur auch nicht gefüllt werden.

SYNTAX: `DrawArc(Pen pen, RectangleF rect, float startwinkel, float bogenwinkel);`



HINWEIS: Positive Winkelangaben werden im Uhrzeigersinn abgetragen.

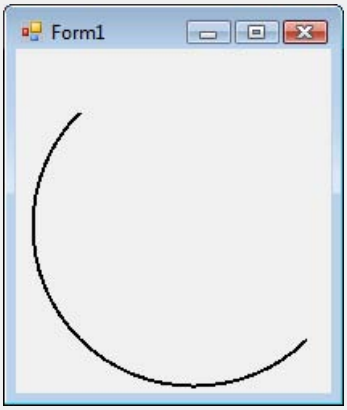
BEISPIEL 33.37: Bogenstück zeichnen

C#

```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 200);

g.DrawArc(new Pen(Color.Black, 2), rec, 45, 180);
```

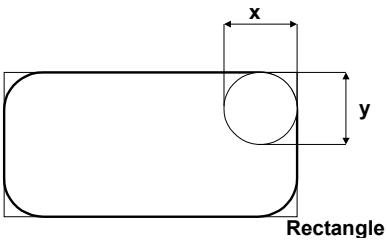
BEISPIEL 33.37: Bogenstück zeichnen

Ergebnis



33.4.13 Wo sind die Rechtecke mit den runden Ecken?

Ersatzlos gestrichen! Es bleibt Ihnen also nichts anderes übrig, als sich eine eigene Funktion zu schreiben, oder Sie nutzen die Möglichkeiten von GDI.



BEISPIEL 33.38: Eine "selbst gestrickte" *RoundRect*-Methode

C#

```
private void RoundRect(Graphics g, Pen p, Rectangle rect, int x, int y)
{
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top, 2 * x, 2 * y), 180, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x, rect.Top, 2 * x, 2 * y),
            270, 90);
  g.DrawArc(p, new Rectangle(rect.Left + rect.Width - 2 * x,
            rect.Top + rect.Height - 2 * y, 2 * x, 2 * y), 0, 90);
  g.DrawArc(p, new Rectangle(rect.Left, rect.Top + rect.Height - 2 * y,
            2 * x, 2 * y), 90, 90);
  g.DrawLine(p, rect.Left + x, rect.Top, rect.Right - x, rect.Top);
  g.DrawLine(p, rect.Left + x, rect.Bottom, rect.Right - x, rect.Bottom);
  g.DrawLine(p, rect.Left, rect.Top + y, rect.Left, rect.Bottom - y);
  g.DrawLine(p, rect.Right, rect.Top + y, rect.Right, rect.Bottom - y);
}
```

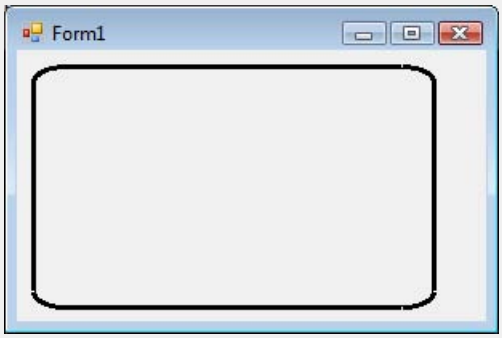
BEISPIEL 33.38: Eine "selbst gestrickte" *RoundRect*-Methode

Beim Aufruf der Methode übergeben Sie ein *Graphics*-Objekt, einen *Pen*, eine *Rectangle*-Struktur für das umgebende Rechteck sowie die Breite und Höhe der "Ecken":

```
Graphics g = this.CreateGraphics();
RoundRect(g, new Pen(Color.Black, 3), new Rectangle(10, 10, 250, 150), 20, 10);
```

Ergebnis

Wie Sie sehen können, ist das Ergebnis schon recht brauchbar:



33.4.14 Textausgabe

Neben den grafischen Primitiven wie Kreis oder Linie wird auch Text unter Windows als Grafik ausgegeben. GDI+ bietet dafür die recht universelle Methode *DrawString*. Leider gibt es reichlich Variationen dieser Methode, was es dem Anfänger sicher nicht einfacher macht. Wir beschränken uns auf zwei wichtige Vertreter:

SYNTAX: `void DrawString(string s, Font font, Brush brush, PointF xy, StringFormat format);`

... Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* an der Stelle *XY*. Wenn notwendig, können Sie weitere Formatierungsanweisungen in *StringFormat* übergeben.

SYNTAX:
```
void DrawString(string s, Font font, Brush brush,
                RectangleF layoutRectangle, StringFormat format);
```

... Ausgabe eines Textes *String* mit der Schrift *Font* und der Füllung *Brush* in einem Rechteck *Rectangle*.

HINWEIS: Mehr Details über das Erstellen von *Font*- bzw. *Brush*-Objekten finden Sie im Abschnitt 33.5.

BEISPIEL 33.39: Ausgabe eines Textes an der Position 70,70

C#

```
Graphics g = this.CreateGraphics();
g.DrawString("Textausgabe", new Font("Arial", 18), new SolidBrush(Color.Black), 70, 70);
```

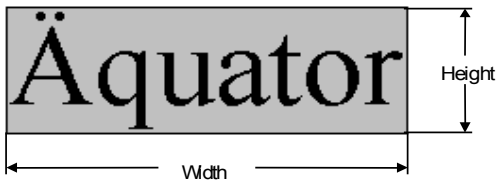
Texteigenschaften

Haben Sie einen String, wie zum Beispiel

```
s = "Rotationswinkel 27""
```

... und möchten Sie diesen **zentriert** ausgeben, brauchen Sie Informationen darüber, wie hoch und wie breit der auszugebende Text ist. Die Methode *MeasureString* ist in diesem Zusammenhang interessant.

Für einen vorgegebenen String mit dem gewählten Font werden Breite und Höhe in einer *SizeF*-Struktur zurückgegeben. Während *SizeF.Height* der Schrifthöhe in der jeweils gewählten Skalierung entspricht, ist *SizeF.Width* von der Anzahl der Buchstaben und der Schriftart abhängig:



Die Ausgabeposition berechnet sich wie folgt:

```
Graphics g = this.CreateGraphics();
Font myfont = new Font("Arial", 36);
SizeF mySize = g.MeasureString("Textausgabe", myfont);
g.DrawString("Textausgabe", myfont, new SolidBrush(Color.Black),
            (this.Width - mySize.Width) / 2, 150);
```

Ausgabe von mehrzeiligem Text

Für die Ausgabe von mehrzeiligem Text nutzen Sie eine überladene Variante von *DrawString*, die zusätzlich eine *Rectangle*-Struktur als Parameter akzeptiert.

HINWEIS: Möchten Sie die Anzahl der entstehenden Zeilen und Spalten ermitteln, können Sie ebenfalls die Methode *MeasureString* nutzen.

BEISPIEL 33.40: Ausgabe von mehrzeiligem Text

C#

```
Graphics g = this.CreateGraphics();
Font myfont = new Font("Arial", 12);
g.DrawString("Textausgabe im Rechteck", myfont, Brushes.Red,
             new RectangleF(10, 10, 60, 180));

g.DrawRectangle(new Pen(Color.Black), new Rectangle(10, 10, 60, 180));
```

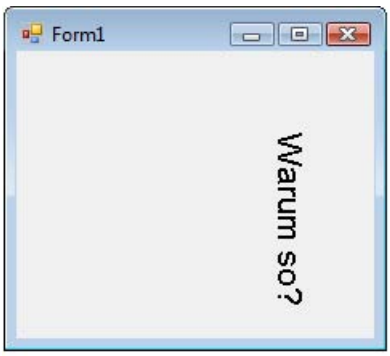
BEISPIEL 33.40: Ausgabe von mehrzeiligem Text

Ergebnis

Textattribute

Über den Parameter *StringFormat* können Sie zusätzliche Formatierungsanweisungen an die *DrawString*-Methode übergeben. Auf die einzelnen Parameter wollen wir an dieser Stelle nicht eingehen, da Sie diese im Normalfall auch kaum gebrauchen werden. Lediglich der Parameter *DirectionVertical* dürfte Ihr Interesse wecken, zu vermuten ist, dass Sie damit einen senkrechten Text ausgeben können.

Sicher, die Autoren haben immer etwas zu "meckern", aber warum *StringFormatFlags.DirectionVertical* einen Text erzeugt, der die folgende Ausrichtung hat, scheint doch etwas merkwürdig[1]:



Ausgabequalität

Wem die Darstellungsqualität des Textes partout nicht passt bzw. wer sich nach einer Antialiasing-Funktion umsieht, wird bei der Eigenschaft *TextRenderingHint* fündig.

BEISPIEL 33.41: Antialiasing einschalten

C#

```
Graphics g = this.CreateGraphics();
g.RotateTransform(15);
```

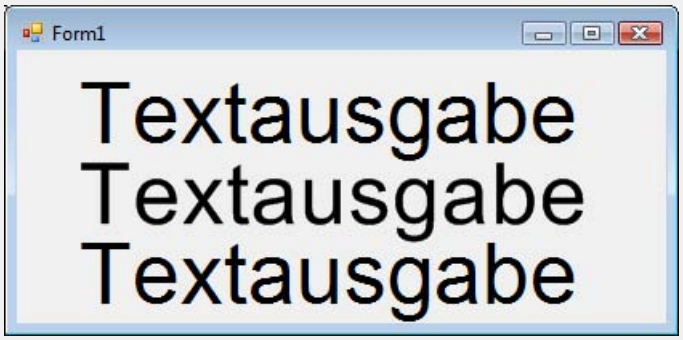
[1] Die Autoren haben bisher kein deutsches Buch oder eine technische Zeichnung gefunden, wo eine derartige Textausrichtung vorkommt. Unsere amerikanischen Freunde hätten es bei der schlichten Angabe eines Drehwinkels belassen sollen.

BEISPIEL 33.41: Antialiasing einschalten

```
g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black), 30, 10);
g.TextRenderingHint = TextRenderingHint.AntiAlias;
g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black), 30, 60);
g.TextRenderingHint = TextRenderingHint.ClearTypeGridFit;
g.DrawString("Textausgabe", new Font("Arial", 40), new SolidBrush(Color.Black), 30, 110);
```

Ergebnis

Das Ergebnis zeigt die folgende Abbildung, die beste Darstellung dürfte der Wert *AntiAlias* liefern.



HINWEIS: Bevor Sie jetzt alle Texte auf diese Weise ausgeben, vergessen Sie bitte nicht, dass dafür auch jede Menge Rechenzeit benötigt wird.

Und wo bleibt eine Methode zum Drehen von Text?

Sie werden auch bei intensivster Suche weder beim *Font*-Objekt noch bei der Methode *DrawString* einen Weg finden, gedrehten Text auszugeben. Wer das Kapitel bis hier aufmerksam gelesen hat, wird sich an die *Graphics*-Methode *RotateTransform* erinnern[1]. Für alle anderen gilt: "Gehe zurück zum Abschnitt 33.3.2"!

HINWEIS: Möchten Sie nacheinander Text mit verschiedenen Winkeln ausgeben, entscheiden Sie mit dem optionalen Parameter *MatrixOrder*, ob die Winkel inkrementell oder absolut angegeben werden.

BEISPIEL 33.42: Rotation von Text um 15°

C#

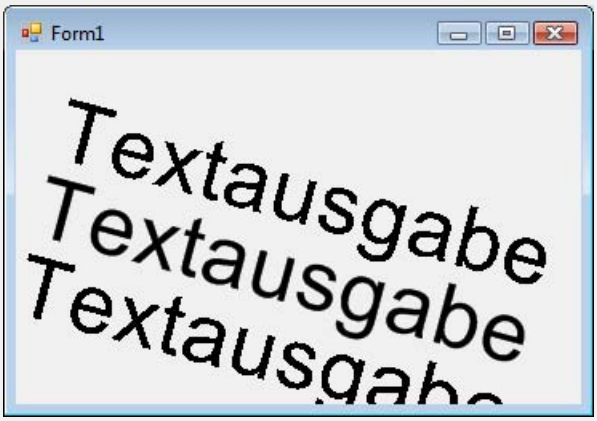
```
Graphics g = this.CreateGraphics();
g.RotateTransform(15);
g.DrawString("Textausgabe", new Font("Arial", 18), new SolidBrush(Color.Black), 70, 70);
```

[1] Diese Form der Umsetzung scheint allerdings doch etwas "sinnfrei" zu sein, berechnen Sie doch bitte einmal schnell die neuen x,y-Koordinaten, wenn Sie einen Text im Winkel von 90 an einem Rechteck ausrichten wollen.

BEISPIEL 33.42: Rotation von Text um 15°

Ergebnis

Kommen wir noch einmal auf unser Beispiel aus dem vorhergehenden Abschnitt zurück. Testen Sie das Beispiel einmal mit gedrehtem Text, werden Sie folgendes Ergebnis erhalten:



BEISPIEL 33.43: Absolute Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Prepend);
```

BEISPIEL 33.44: Inkrementelle Angabe

C#

```
g.RotateTransform(10, System.Drawing.Drawing2D.MatrixOrder.Append);
```

33.4.15 Ausgabe von Grafiken

Unter "Ausgabe von Grafiken" möchten wir an dieser Stelle die Wiedergabe von fertigen Grafiken (Bitmaps, Icons, Metafiles) auf einem *Graphics*-Objekt verstehen.

SYNTAX: `void DrawImageUnscaled(Image image, int x, int y);`

... zeichnet die angegebene Grafik an der durch x und y angegebenen Position.

HINWEIS: Die anderen Varianten von *DrawImageUnscaled* können Sie getrost vergessen, trotz angekündigtem Beschneidens der Grafik erfolgt dies nicht.

BEISPIEL 33.45: Verwendung von *DrawImageUnscaled*

C#

```
Graphics g = this.CreateGraphics();
g.DrawImageUnscaled(pictureBox1.Image, 0, 0);
```

Skalieren

Für die skalierte Ausgabe von Grafiken können Sie *DrawImage* verwenden:

```
SYNTAX: void DrawImage(Image image, int x, int y, int width, int height);
```

```
SYNTAX: void DrawImage(Image image, int x, int y, Rectangle srcRect, GraphicsUnit srcUnit);
```

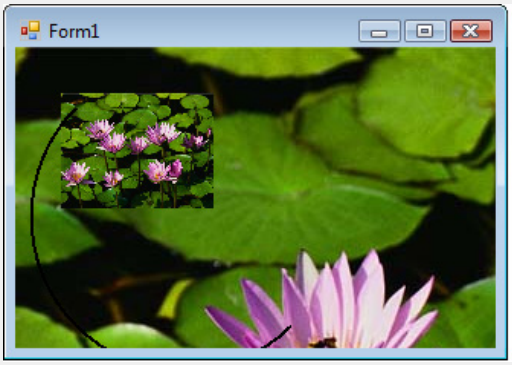
Während die erste Variante neben der Position auch die Breite und Höhe erwartet, können Sie bei der zweiten Syntax-Variante zusätzlich die Einheiten (*GraphicsUnit*) angeben (z.B. beim Drucker).

BEISPIEL 33.46: Verkleinerte, aber proportionale Ausgabe eines Image

C#

```
Single faktor;
Graphics g = this.CreateGraphics();
faktor = PictureBox1.Image.Height / PictureBox1.Image.Width;
g.DrawImageUnscaled(PictureBox1.Image, 0, 0);
g.DrawImage(PictureBox1.Image, 30, 30, 100, 100 * faktor);
```

Ergebnis



33.5 Unser Werkzeugkasten

Nachdem Sie sich im vorhergehenden Abschnitt mit diversen Grafikmethoden herumgeschlagen haben, wollen wir Ihnen jetzt die einzelnen "Werkzeuge" des Grafikprogrammierers sowie deren Konfigurationsmöglichkeiten näher vorstellen.

33.5.1 Einfache Objekte

Zunächst sollten wir kurz auf einige grundlegende Strukturen im Zusammenhang mit der Grafikausgabe eingehen.

Point, FPoint

Für die Angabe von Koordinaten wird bei vielen Grafikmethoden ein *Point*-Objekt (Integer) bzw. ein *PointF* (Gleitkommawert) verwendet. Die beiden wichtigsten Eigenschaften: x, y.

BEISPIEL 33.47: Deklaration eines neuen *Point*-Objekts (x = 10, y = 10)

C#
```
Point p = new Point(10, 10);
```

BEISPIEL 33.48: Direktes Verändern der x-Koordinate

C#
```
Point p = new Point(10, 10);
p.X = 100;
```

BEISPIEL 33.49: Verschieben des Punktes

C#
```
Point p = new Point(10, 10);
p.Offset(10, 10);
```

BEISPIEL 33.50: Vergleich von zwei Punktkoordinaten

C#
```
Point p1 = new Point(10, 10);
Point p2 = new Point(10, 10);
if (p1.Equals(p2)) MessageBox.Show("Punkte sind gleich");
```

BEISPIEL 33.51: Konvertieren in einen Gleitkomma-Point (PointF):

C#
```
Point p1 = new Point(10, 10);
PointF p2;
p2 = Point.op_Implicit(p1);          // Variante 1
p2 = (PointF) p1;                    // Variante 2
```

Size, FSize

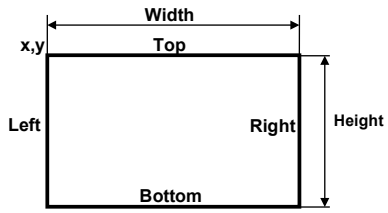
Ähnlich wie *Point* verwaltet *Size* bzw. *SizeF* ein Koordinatenpaar, nur dass es sich in diesem Fall um die Breite (*Width*) bzw. Höhe (*Height*) handelt.

BEISPIEL 33.52: Deklarieren und Verändern der Breite

C#
```
Size mysize = new Size(100, 80);
mysize.Width = 130;
```

Rectangle, FRectangle

Das dritte Grundobjekt ist im Grunde die Kombination der beiden vorhergehenden Objekte. Die folgende Skizze zeigt die verschiedenen Formen des Zugriffs auf die Koordinatenpaare:



BEISPIEL 33.53: Deklarieren und verwenden von *Rectangle*

C#
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 100);
g.FillRectangle(new SolidBrush(Color.Red), rec);
```

33.5.2 Vordefinierte Objekte

Vielleicht hatte irgendein Programmierer bei Microsoft ein Einsehen und entschied sich dafür, wenigstens einige Grafikobjekte vorzudefinieren, d h., ohne dass Sie diese erst mit viel Aufwand und umfangreichen Parameterlisten initialisieren müssen.

Drei Gruppen können Sie unterscheiden:

- vordefinierte Pinsel (Brushes)
- vordefinierte Stifte (Pens)
- vordefinierte Farben (Colors)
- vordefinierte Grafiken (Icons)

Vordefinierte Pinsel

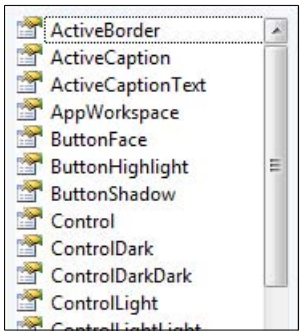
Neben *Brushes* (alle relevanten Farben) können Sie auch *SystemBrushes* (die vordefinierten Systemfarben) verwenden.

BEISPIEL 33.54: Erzeugt einer leicht gelb getönte Fläche (Farbe der Hints).

C#
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 100);
g.FillRectangle(SystemBrushes.Info, rec);
```

Vordefinierte Stifte

Über das *SystemPens*-Objekt bzw. dessen Eigenschaften rufen Sie vordefinierte Stifte (Breite = 1 Pixel) für alle Windows-Farben ab:



BEISPIEL 33.55: Verwendung eines System-Pens

C#
```
Graphics g = this.CreateGraphics();
g.DrawRectangle(SystemPens.ControlText, 10, 10, 100, 100);
```

Vordefinierte Farben

Farben können Sie entweder über *SystemColors* (die Farben der Windows-Elemente) oder über *Color* abrufen.

BEISPIEL 33.56: Verwendung von *Color*

C#
```
Graphics g = this.CreateGraphics();
Rectangle rec = new Rectangle(10, 10, 200, 200);
g.FillPie(new SolidBrush(Color.Red), rec, 0, 100);
```

Vordefinierte Icons

Über *SystemIcons* können Sie die wichtigsten System-Icons direkt abrufen[1]:

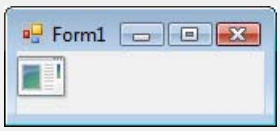
Eigenschaft
Application
Asterisk
Error
Exclamation
Hand
Information
Question
Warning
WinLogo

BEISPIEL 33.57: Verwendung eines System-Icons

C#
```
Graphics g = this.CreateGraphics();
g.DrawIcon(SystemIcons.Application, 0, 0);
```

Ergebnis

Die Grafikausgabe:



[1] Diese unterscheiden sich etwas je nach System, deshalb hier keine Abbildungen.

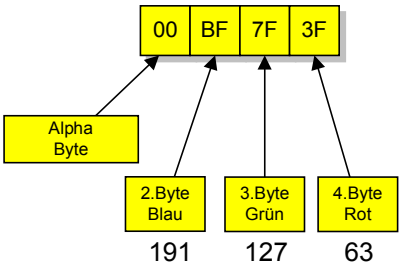
33.5.3 Farben/Transparenz

Farben setzen sich auch unter Windows aus der additiven Überlagerung der drei Grundfarben Rot, Grün und Blau zusammen (RGB). Da jeder Farbanteil in 256 Farbstufen unterteilt ist, ergibt sich eine maximale Anzahl von ca. 16 Mio. Farben. Bei dieser Auflösung ist das menschliche Auge nicht mehr in der Lage, einzelne Abstufungen wahrzunehmen, man spricht von Echtfarben.

ARGB-Farben

GDI+ verwendet für die Verwaltung von Farbinformationen so genannte ARGB-Werte, d h. 4 Byte- bzw. Integer-Werte. Auch die bereits vordefinierten Farbwerte (siehe vorhergehender Abschnitt) verwenden dieses Farbmodell.

Um zu verstehen, wie die Farbwerte gespeichert werden können, müssen wir wissen, dass zur Darstellung **eines** Bytes **zwei** Hexziffern benötigt werden. Eine einzelne Hexziffer wird durch eines der 16 Zeichen 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F dargestellt. Die drei niederwertigen Bytes geben die RGB-Farbintensität für Blau, Grün und Rot an.



BEISPIEL 33.58: Farbwerte

C#

Da pro Byte 256 Werte gespeichert werden können, entspricht der Wert 0x00FF0000 einem reinen Blau mit voller Intensität, der Wert 0x0000FF00 einem reinen Grün und der Wert 0x000000FF einem reinen Rot. 0x00000000 gibt Schwarz und 0x00FFFFFF Weiß an. In obiger Abbildung wird eine graublaue Farbe definiert.

HINWEIS: Möchten Sie einen Graustufenwert erzeugen, müssen die einzelnen Farbanteile jeweils den gleichen Wert aufweisen.

Was ist mit dem höchstwertigen Byte?

Bei ARGB-Werten wird in diesem Byte die Information über die Transparenz dieser Farbe gespeichert. Ein Wert von 255 entspricht der vollen Deckkraft, 0 entspricht vollständiger Transparenz. Mit Hilfe der Methode *FromArgb* können Sie einen beliebigen Farbwert mit der gewünschten Transparenz erzeugen.

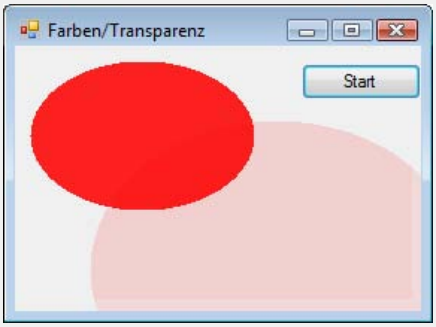
BEISPIEL 33.59: Erzeugen von zwei teiltransparenten Ellipsen

C#

```
Graphics g  = this.CreateGraphics();
Color c1, c2;
c1 = Color.FromArgb(125, Color.Red);
g.FillEllipse(new SolidBrush(c1), new Rectangle(10, 10, 150, 100));
c2 = Color.FromArgb(12, Color.Red);
g.FillEllipse(new SolidBrush(c2), new Rectangle(50, 50, 250, 200));
```

Ergebnis

Das Ergebnis auf dem Formular:



Rufen Sie obige Anweisungen mehrfach auf, addieren sich die Farbwerte immer weiter, bis eine vollständige Deckung erreicht wird.

Möchten Sie zwei Farbwerte vergleichen, verwenden Sie die Methode *ToArgb*:

```
if (c1.ToArgb() == c2.ToArgb()) MessageBox.Show("Beide Farben sind gleich!");
```

33.5.4 Stifte (Pen)

Für Grafikmethoden, die Linien verwenden, benötigen Sie ein initialisiertes *Pen*-Objekt. Dieses enthält unter anderem Informationen über:

- die Farbe
- die Dicke
- die Linienenden
- die Verbindung zweier zusammenhängender Linien
- den Füllstil

Bevor wir Sie mit einer Fülle von Eigenschaften erschlagen, sollen einige Beispiele für mehr Klarheit sorgen.

Einfarbige Stifte

Einfarbige Stifte werden mit einer der vordefinierten Farben oder einem ARGB-Wert als Parameter erzeugt.

BEISPIEL 33.60: Erzeugen eines roten Stifts

C#
```
Pen myPen = new Pen(Color.Red);
```

BEISPIEL 33.61: Erzeugen eines einfarbigen Pens mit 50 % Transparenz und 10 Pixeln Breite

C#
```
Pen myPen = new Pen(Color.FromArgb(128, 17, 69, 137), 10);
```

BEISPIEL 33.62: Zeichnen von zwei aufeinander folgenden roten Linien (15 Pixel breit) mit einem Pfeil am Anfang und einem runden Ende

C#
```
Graphics g  = this.CreateGraphics();
```

Zunächst einen einfarbigen *Pen* der Stärke 15 erzeugen:

```
Pen myPen = new Pen(Color.Red, 15);
```

Die Punkte für die Linienenden festlegen:

```
PointF[] punkte = new PointF[] {new PointF(10, 10), new PointF(100, 100),
                                new PointF(50, 150)};
```

Den Linienstart festlegen (Pfeil):

```
  myPen.EndCap = LineCap.Round;
```

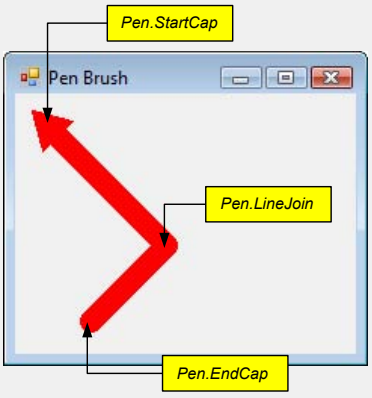
Das Linienende festlegen (rund):

```
  myPen.StartCap = LineCap.ArrowAnchor;
```

Die Verbindung der zwei Teillinien festlegen (rund) und Linien zeichnen:

```
  myPen.LineJoin = LineJoin.Round;
  g.DrawLines(myPen, punkte);
```

Ergebnis



Einige wichtige Linieneigenschaften auf einen Blick:

Eigenschaft	Beschreibung
Alignment	... beschreibt die Position (*Center, Inset, Outset, Left, Right*) der Linie bezüglich der gedachten Ideallinie zwischen zwei Punkten.
Color	... die Füllfarbe des Stifts.
LineJoin	... beschreibt den Übergang (*Bevel, Miter, MiterClipped, Round*) zwischen zwei aufeinander folgenden Linien. Siehe dazu vorhergehendes Beispiel.
StartCap, EndCap	... beschreibt die Form der Linienenden (*Flat, Round, Square ...*). Siehe dazu vorhergehendes Beispiel.
DashCap	... beschreibt die Form der einzelnen Punkte/Linienabschnitte bei gestrichelten Linien (*Flat, Round Triangle*).
DashStyle	... beschreibt die Form von gestrichelten Linien (*Custom, Dash, DashDot, DashDotDot, Dot, Solid*).
PenType	... beschreibt die Stiftart (*SolidColor, Hatchfill, LinearGradient, PathGradient, TextureFill*). Diese Eigenschaft ist schreibgeschützt.

Stifte mit Füllung

Im Unterschied zu den bisher vorgestellten Stiften, die lediglich unterschiedliche Linienmuster bzw. -farben aufweisen konnten, lassen sich auch Stifte erzeugen, die mit einem *Brush* statt mit einer Farbe initialisiert werden.

BEISPIEL 33.63: Erzeugen eines Stifts mit Brush als Füllung

C#

```
Graphics g = this.CreateGraphics();
Pen mypen = new Pen(brushes.Azure, 10);

g.DrawLine(mypen, 10, 10, 100, 100);
```

Auf den ersten Blick werden Sie keinen Unterschied zu den bisherigen Pens erkennen, da wir einen *SolidBrush* verwendet haben.

Etwas anders sieht die Linie allerdings aus, wenn Sie zum Beispiel einen *HatchBrush* (siehe auch folgender Abschnitt) für die Linienfüllung verwenden.

BEISPIEL 33.64: *HatchBrush* für Linie erzeugen

C#

Vergessen Sie nicht, den entsprechenden Namespace einzubinden:

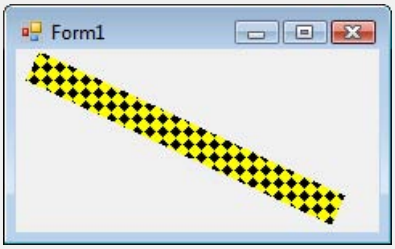
```
using System.Drawing.Drawing2D;
...
```

BEISPIEL 33.64: *HatchBrush* für Linie erzeugen

```
Graphics g = this.CreateGraphics();
HatchBrush myBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.Black, Color.Yellow);
Pen myPen = new Pen(myBrush, 20);
g.DrawLine(myPen, 10, 10, 200, 100);
```

Ergebnis

Das Ergebnis unterscheidet sich doch wesentlich von den bisher bekannten Linienarten:



Wie Sie sehen, können Sie fast beliebige Stiftfüllungen erzeugen, Sie müssen nur einen entsprechenden Brush (siehe folgender Abschnitt) zur Verfügung stellen.

HINWEIS: Verwechseln Sie nicht Stifte mit Pinseln. Mit Stiften können Sie niemals eine Figur füllen, mit Pinseln können Sie keine Figuren zeichnen, sondern nur füllen.

33.5.5 Pinsel (Brush)

Damit sind wir bei einem der wohl komplexesten Zeichenobjekte angelangt. Neben dem bereits mehrfach in diesem Kapitel verwendeten *SolidBrush* und dem im vorhergehenden Abschnitt bereits angesprochenen *HatchBrush* finden Sie noch weitere drei Vertreter dieser Gattung.

Alle Pinsel auf einen Blick:

Typ	Beschreibung
SolidBrush	... ein Pinsel mit einheitlicher Farbe und ohne Muster.
HatchBrush	... ein Pinsel mit einem Linienmuster sowie Vordergrund- (Linie) und Hintergrundfarbe.
TextureBrush	... ein Pinsel mit Images/Bitmaps als Füllmuster.
LinearGradientBrush	... ein Pinsel mit einem Farbverlauf als Füllung (Verlauf zwischen zwei Farben).
PathGradientBrush	... ein Pinsel mit mehreren Farbverläufen (Verlauf zwischen einer Farbe und mehreren anderen).

Gefüllte Objekte, basierend auf den oben genannten Pinseln, erzeugen Sie mit den entsprechenden *Fill*xyz-Methoden (z.b: *FillRectangle*) des *Graphics*-Objekts.

33.5.6 SolidBrush

Das Erzeugen eines *SolidBrush* dürfte Sie kaum vor Probleme stellen, übergeben Sie eine Farbe oder verwenden Sie die Methode *FromArgb*, um eine neue Farbe zu definieren.

BEISPIEL 33.65: Roter Pinsel

C#
```
SolidBrush mybrush = new SolidBrush(Color.Red);
```

BEISPIEL 33.66: Teilweise transparenter Pinsel mit selbst definierter Farbe

C#
```
SolidBrush mybrush = new SolidBrush(Color.FromArgb(123, 10, 17, 36));
```

33.5.7 HatchBrush

Bevor Sie sich näher mit dieser Pinselvariante beschäftigen können, müssen Sie den notwendigen Namespace *System.Drawing.Drawing2D* einbinden.

Ein *HatchBrush* besteht aus einem Vordergrundmuster mit eigener Farbe und einer Hintergrundfarbe. Der Konstruktor:

SYNTAX: `HatchBrush(HatchStyle hatchstyle, Color foreColor, Color backColor);`

Übergeben können Sie unter anderem folgende Konstanten:

HatchStyle
BackwardDiagonal
Cross
DarkDownwardDiagonal
DarkHorizontal
DarkUpwardDiagonal
DarkVertical
DashedDownwardDiagonal
DashedHorizontal
DashedUpwardDiagonal
DashedVertical
DiagonalBrick
DiagonalCross
Divot
DottedDiamond
DottedGrid
ForwardDiagonal

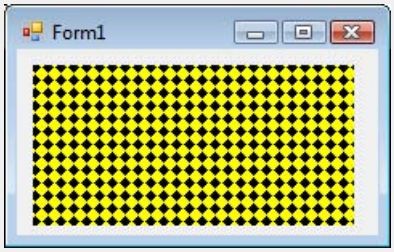
BEISPIEL 33.67: Erzeugen und verwenden einer *HatchBrush*

C#

```
Graphics g = this.CreateGraphics();
HatchBrush myBrush = new HatchBrush(HatchStyle.SolidDiamond, Color.Black, Color.Yellow);
g.FillRectangle(myBrush, 10, 10, 200, 100);
```

Ergebnis

Das erzeugte Rechteck:



HINWEIS: Eine Skalierung des Ausgabegeräts (*ScaleTransform*) hat keinen Einfluss auf die Mustergröße.

33.5.8 TextureBrush

Mit einem *TextureBrush* lassen sich Flächen mit Bitmap-Mustern füllen. Ist die Bitmap zu klein, um die gesamte Fläche auszufüllen, wird diese wiederholt dargestellt.

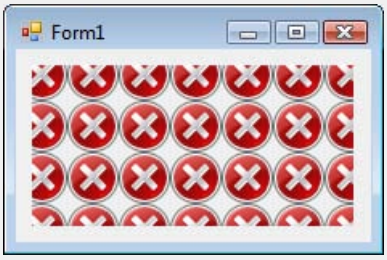
BEISPIEL 33.68: Basierend auf den System-Icons soll ein Rechteck mit Error-Symbolen gefüllt werden.

C#

```
Graphics g = this.CreateGraphics();
TextureBrush myBrush = new TextureBrush(SystemIcons.Error.ToBitmap());
g.FillRectangle(myBrush, 10, 10, 200, 100);
```

Ergebnis

Die Ausgabe:



Natürlich stellt es auch kein Problem dar, die Grafik aus einer Datei zu laden.

BEISPIEL 33.69: Laden der Datei *Bitmap1.bmp* als *TextureBrush*

C#
```
Graphics g = this.CreateGraphics();
Image img  = new Bitmap("bitmap1.bmp");

TextureBrush myBrush = new TextureBrush(img);
g.FillRectangle(myBrush, 10, 10, 200, 100);
```

33.5.9 LinearGradientBrush

Mit dem *LinearGradientBrush* bringen Sie im wahrsten Sinne des Wortes mehr Farbe in Ihre Anwendungen. Das Grundprinzip: Sie geben zwei Farben und eine Richtung (daher das "linear") an, und GDI+ berechnet Ihnen den zugehörigen Farbverlauf.

An den Konstruktor müssen Sie folgende Werte übergeben:

SYNTAX:
```
LinearGradientBrush(Rectangle rect, Color startfarbe, Color endfarbe,
                    LinearGradientMode linearGradientMode);
```

Rectangle gibt ein Rechteck an (es sind auch zwei *Point*-Werte zulässig), in dem der Farbverlauf berechnet wird. Die Betonung liegt auf "berechnet", welche Ausgabefläche Sie später mit dem neuen *Brush* füllen, ist eine ganz andere Frage.

Start- und Endfarbe sind normale ARGB-Color-Werte mit Transparenzangabe, d.h., wenn Sie beispielsweise zwei gleiche Farben, aber unterschiedliche Alpha-Werte angeben, können Sie einen Farbverlauf mit zu- bzw. abnehmender Transparenz realisieren.

Folgende *LinearGradientMode*-Werte sind zulässig:

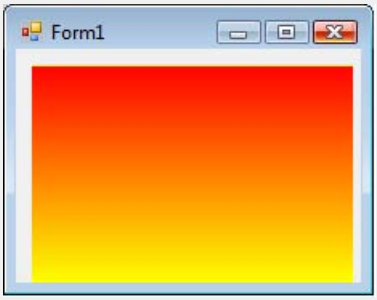
Konstante	Beschreibung
BackwardDiagonal	Farbverlauf von rechts oben nach links unten
ForwardDiagonal	Farbverlauf von links oben nach rechts unten
Horizontal	Farbverlauf von links nach rechts
Vertical	Farbverlauf von oben nach unten

BEISPIEL 33.70: Ein vertikaler Farbverlauf von Rot nach Gelb soll realisiert werden

C#
```
Graphics g = this.CreateGraphics();
Rectangle rect = new Rectangle(10, 10, 200, 135);
LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);
g.FillRectangle(myBrush, new Rectangle(10, 10, 200, 185));
```

BEISPIEL 33.70: Ein vertikaler Farbverlauf von Rot nach Gelb soll realisiert werden

Ergebnis



Wer ganz genau hinsieht, wird allerdings schnell "ein Haar in der Suppe" finden. Die erste Zeile des Ausgaberechtecks ist nicht rot, sondern gelb. Die Ursache ist vermutlich die Wiederholung des Farbverlaufs, wenn der berechnete Verlauf kleiner als die Ausgabefläche ist.

Nehmen Sie folgende Änderung vor, um die gelbe Zeile zu entfernen:

```
Rectangle rect = new Rectangle(10, 10, 200, 185);
LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);

                g.FillRectangle(myBrush, new Rectangle(10, 10, 200, 180));
```

33.5.10 PathGradientBrush

Hier haben wir es mit einer Spezialform von Gradienten zu tun, die es uns erlauben, recht eindrucksvolle Farbeffekte zu realisieren. Ausgehend von einer zentralen Farbe können Sie innerhalb eines *Path*-Objekts (siehe dazu Abschnitt 33.5.12) mehrere Farbverläufe zu verschiedenen Farben realisieren.

BEISPIEL 33.71: Erzeugen und verwenden eines *PathGradientBrush*

C#

```
        Graphics g = this.CreateGraphics();
```

Zunächst erzeugen wir das *Path*-Objekt:

```
        GraphicsPath path = new GraphicsPath();
        path.AddLine(10, 10, 300, 10);
        path.AddLine(300, 10, 250, 200);
```

BEISPIEL 33.71: Erzeugen und verwenden eines *PathGradientBrush*

```
path.AddLine(250, 200, 150, 250);
path.AddLine(150, 250, 50, 200);
```

Jetzt können wir mit diesem *Path* den *PathGradientBrush* erzeugen und konfigurieren:

```
PathGradientBrush myBrush = new PathGradientBrush(path);
```

Ein Array mit den Farben für die jeweiligen Eckpunkte im *Path*:

```
Color[] myColors = new Color[] {Color.Yellow, Color.Green, Color.Red,
                                Color.Cyan, Color.Blue};
```

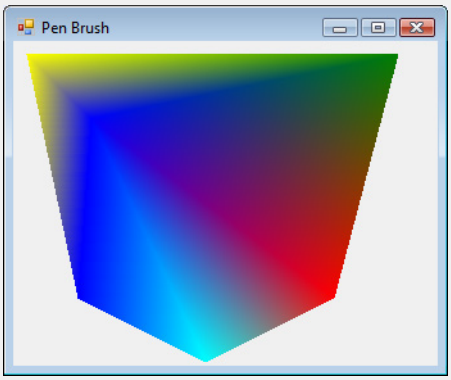
Die Farbe im Mittelpunkt[1] und an den Ecken des *Path*:

```
myBrush.CenterColor = Color.Blue;
myBrush.SurroundColors = myColors;
```

Wir geben den *Path* aus:

```
myBrush.CenterPoint = new PointF(60, 60);
g.FillPath(myBrush, path);
```

Ergebnis



33.5.11 Fonts

Wie bei Stiften und Pinseln handelt es sich auch bei Schriften (Fonts) um Objekte, die Sie zunächst erzeugen müssen. Sage und schreibe dreizehn verschiedene Konstruktorüberladungen werden Ihnen angeboten, wir belassen es bei den beiden folgenden Varianten:

SYNTAX: `Font(string familyName, float emSize);`

SYNTAX: `Font(string familyName, float emSize, FontStyle style);`

[1] "Mittelpunkt" sollten Sie nicht zu genau nehmen, über die Eigenschaft *CenterPoint* können Sie den Mittelpunkt selbst definieren.

Was mit *familyName* und *emSize* gemeint ist, dürfte klar sein, über *style* können Sie eine Kombination der folgenden *FontStyle*-Werte zuweisen:

- *Bold* (Fett)
- *Italic* (Kursiv)
- *Regular* (Normal)
- *Strikeout* (Durchgestrichen)
- *Underline* (Unterstrichen)

Kombinieren können Sie die Werte durch die Verknüpfung mit dem OR-Operator.

HINWEIS: Die Farbe bzw. die Füllung der Schriftart legen Sie erst bei der Ausgabe mittels *DrawString* über einen entsprechenden *Brush* fest. Damit können Sie auch Farbverläufe oder Transparenzeffekte bei Fonts erreichen.

Genug der Theorie, ein praktisches Beispiel soll die Verwendung zeigen.

BEISPIEL 33.72: Erzeugen und verwenden eines neuen Font-Objekts

C#

```
Graphics g = this.CreateGraphics();
Font f = new Font("Arial", 24, FontStyle.Italic | FontStyle.Bold);
g.DrawString("Test Schriftarten", f, new SolidBrush(Color.Blue), 10, 10);
```

Ergebnis

Ein Blick auf die Eigenschaften des *Font*-Objekts macht uns neugierig: Mit *Bold, Italic, Size* etc. stehen uns alle Möglichkeiten zur Konfiguration der Schriftart zur Verfügung. Doch ach: Alle wünschenswerten Eigenschaften sind schreibgeschützt, Sie müssen also wohl oder übel eine neue Schriftart erzeugen.

33.5.12 Path-Objekt

Path-Objekte fassen mehrere Zeichenoperationen (*DrawLine, DrawString, DrawEllipse* ...) bzw. mehrere grafische Primitive (Linien, Kreise etc.) quasi in einem Objekt zusammen. Sie können *Path*s am besten mit der Gruppieren-Funktion eines Grafikprogramms vergleichen. Ähnlich wie mit der Gruppe können Sie den kompletten *Path* mit einer Anweisung auf einem *Graphics*-Objekt wiedergeben (*DrawPath* oder *FillPath*). Welche Linientypen bzw. welcher Füllstil benutzt wird, entscheiden Sie erst beim Zeichnen auf dem *Graphics*-Objekt, nicht beim Erstellen des Objekts.

BEISPIEL 33.73: Erzeugen eines einfachen *Path* (nur zwei Ellipsen und zwei Linien) und Wiedergabe auf dem *Form*-Objekt

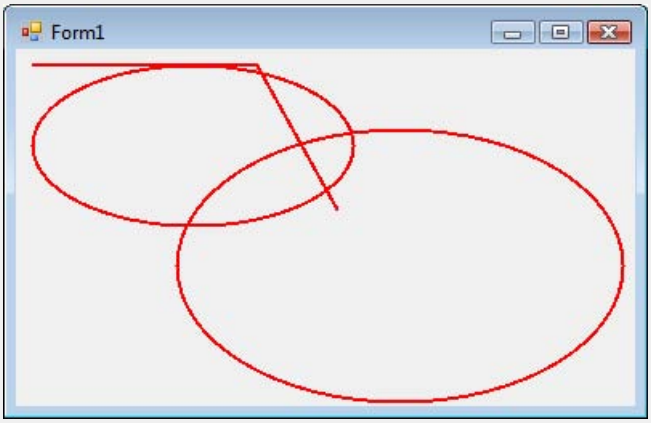
C#

```
using System.Drawing.Drawing2D;
...
   Graphics g = this.CreateGraphics();
   GraphicsPath myPath = new GraphicsPath();

   myPath.AddEllipse(10, 10, 200, 100);
   myPath.AddEllipse(100, 50, 278, 170);
   myPath.AddLine(10, 10, 150, 10);
   myPath.AddLine(150, 10, 200, 100);
```

Wie Sie sehen, wird erst bei der Wiedergabe des *Path*-Objekts ein entsprechender *Pen* zugeordnet:

```
   g.DrawPath(new Pen(Color.Red, 2), myPath);
```

Ergebnis



Füllen

Sie können ein beliebig zusammengesetztes *Path*-Objekt mit einem Pinsel Ihrer Wahl füllen lassen. Ein Beispiel hatten Sie ja bereits im Abschnitt 33.5.10 (PathGradientBrush) kennen gelernt. Das Grundprinzip ist immer gleich. Sie erzeugen ein *Path*-Objekt, fügen diesem die gewünschten Grafikanweisungen hinzu und geben dann diesen *Path* auf einem *Graphics*-Objekt aus.

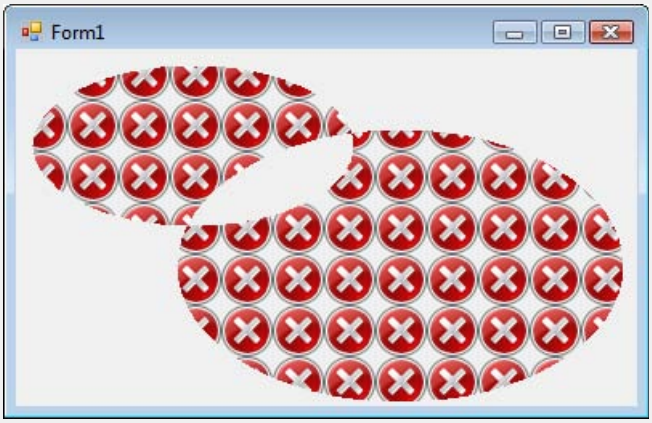
BEISPIEL 33.74: Füllen mit einem Bitmap-Muster

C#

```
   Graphics g = this.CreateGraphics();
   GraphicsPath myPath = new GraphicsPath();
   TextureBrush myBrush = new TextureBrush(SystemIcons.Error.ToBitmap());
   myPath.AddEllipse(10, 10, 200, 100);
   myPath.AddEllipse(100, 50, 278, 170);
   g.FillPath(myBrush, myPath);
```

BEISPIEL 33.74: Füllen mit einem Bitmap-Muster

Ergebnis

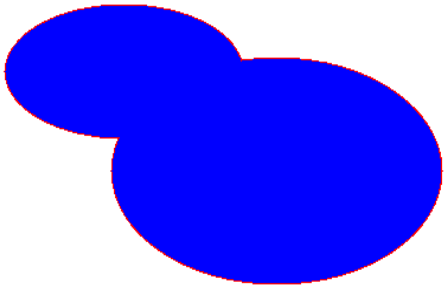


Fillmode

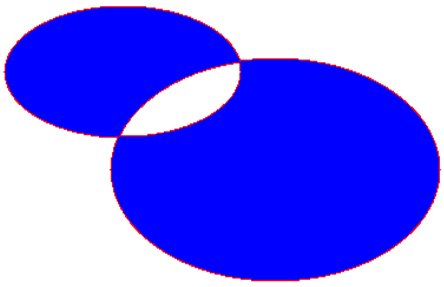
In diesem Zusammenhang ist Ihnen sicher aufgefallen, dass der sich überschneidende Bereich beider Ellipsen nicht gefüllt worden ist.

Wie überlappende bzw. sich schneidende Objekte gefüllt werden, bestimmen Sie mit der Eigenschaft *FillMode*. Statt vieler Worte über die Funktionsweise dürften die beiden folgenden Abbildungen mehr über die Funktionsweise aussagen:

FillMode = FillMode.Winding



FillMode = FillMode.Alternate



Sehen wir uns mit diesem Wissen noch einmal unser Einstiegsbeispiel an, verwenden jedoch statt der *DrawPath*- die *FillPath*-Methode.

BEISPIEL 33.75: Füllen eines Paths

C#

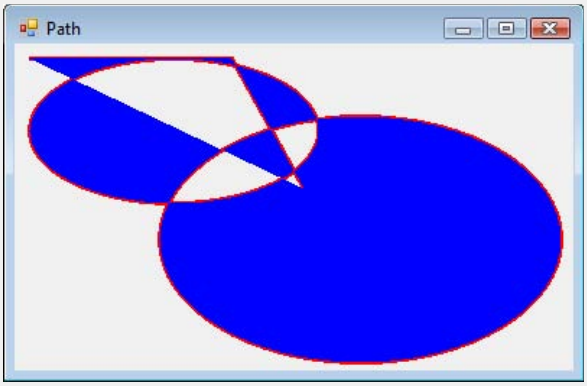
```
Graphics g = this.CreateGraphics();
GraphicsPath myPath = new GraphicsPath();
myPath.AddEllipse(10, 10, 200, 100);
myPath.AddEllipse(100, 50, 278, 170);
myPath.AddLine(10, 10, 150, 10);
myPath.AddLine(150, 10, 200, 100);
myPath.CloseAllFigures();
```

BEISPIEL 33.75: Füllen eines Paths

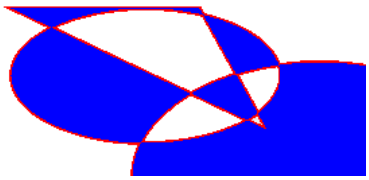
```
g.DrawPath(new Pen(Color.Red, 3), myPath);
g.FillPath(Brushes.Blue, myPath);
```

Ergebnis

Im Ergebnis werden Sie feststellen, dass beim Füllen auch die beiden Linien eine Rolle spielen.



Zwischen den beiden Endpunkten der Linien wird zwar keine Linie gezeichnet, eine Füllung kommt unter Berücksichtigung von *Fillmode* jedoch zustande. Soll aus den beiden Linien eine in sich geschlossene Fläche erzeugt werden, rufen Sie die Methode *CloseAllFigures* auf, die die beiden Linien zu einem Dreieck verbindet (achten Sie auf die neue Verbindungslinie zwischen den beiden Linienendpunkten):



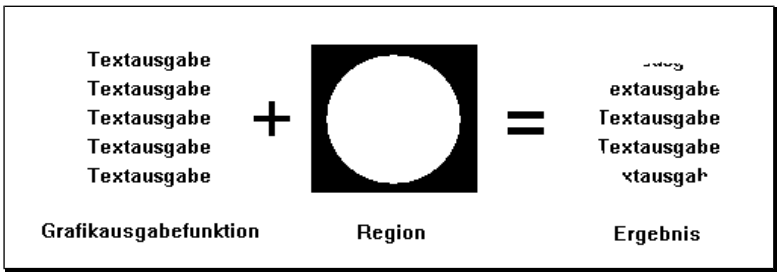
HINWEIS: Ein weiteres Einsatzgebiet für *Path*-Objekte findet sich im Zusammenhang mit dem Clipping, das wir im folgenden Abschnitt näher betrachten wollen.

33.5.13 Clipping/Region

In Ihren Programmen sind Sie es gewohnt, dass Zeichenoperationen, die über den Clientbereich des Formulars bzw. der Komponente hinausgehen, einfach abgeschnitten werden. Damit haben Sie auch schon die einfachste Form von Clipping kennen gelernt.

Für bestimmte visuelle Effekte ist es häufig wünschenswert, dass diese Ausgabebereiche nicht nur rechteckig, sondern zum Beispiel auch mal rund oder aus verschiedenen grafischen Primitiven zusammengesetzt sind.

Die Abbildung zeigt das Ergebnis von Grafikausgaben in einem runden Clipping-Bereich:



Ein Clipping-Bereich ist immer einem *Graphics*-Objekt zugewiesen, mit Hilfe der Methode *SetClip* können Sie diesen Bereich verändern.

Unter GDI+ bieten sich mehrere Varianten an:

- Übernahme von anderem *Graphics*-Objekt (*Graphics.SetClip(Graphics)*)
- Clipping in einem Rechteck (*Graphics.SetClip(Rectangle)*)
- Clipping in einem *Path* (*Graphics.SetClip(GraphicsPath)*)
- Clipping in einer *Region* (*Graphics.SetClip(Region)*)

Im Folgenden beschränken wir uns auf die letzte Variante, die Vorgehensweise ist auch bei einem *Path* oder einem Rechteck immer gleich.

Regions

Regions können, im Zusammenhang mit dem Clipping, quasi wie eine Schablone betrachtet werden. Da Sie *Regions* recht einfach zusammensetzen können (mit unterschiedlichen Verknüpfungen), dürften sie die erste Wahl für die meisten Clipping-Aufgaben sein.

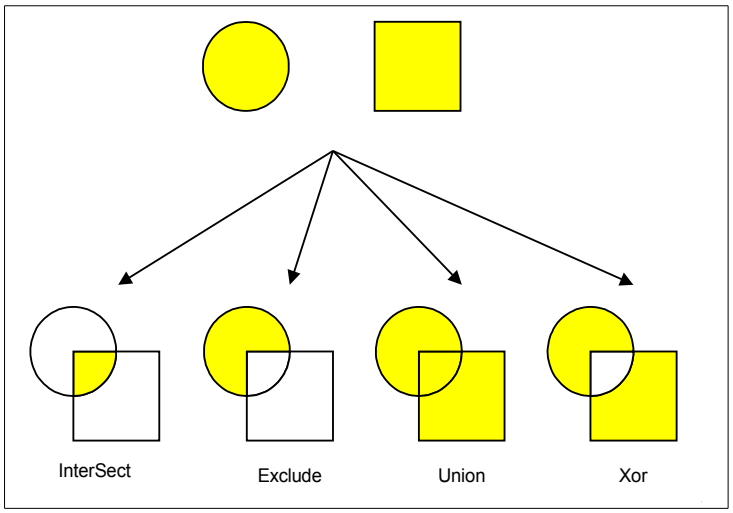
Das Erzeugen einer Region ist relativ simpel. Über den Konstruktor des *Region*-Objekts können Sie bereits entscheiden, worauf die Region aufgebaut wird:

- ein *Path*-Objekt
- ein Rechteck
- ein anderes *Region*-Objekt

BEISPIEL 33.76: Erzeugen und füllen einer rechteckigen Region

C#
```
using System.Drawing.Drawing2D;
...
 Graphics g = this.CreateGraphics();
 SolidBrush blueBrush = new SolidBrush(Color.Blue);
 Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
 g.FillRegion(blueBrush, Reg1);
```

Zwei Regionen können Sie mit Hilfe der *Region*-Methoden auch kombinieren:



BEISPIEL 33.77: Kombination zweier rechteckiger Regionen

C#

```
Graphics g = this.CreateGraphics();

SolidBrush blueBrush = new SolidBrush(Color.Blue);
Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
Region Reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

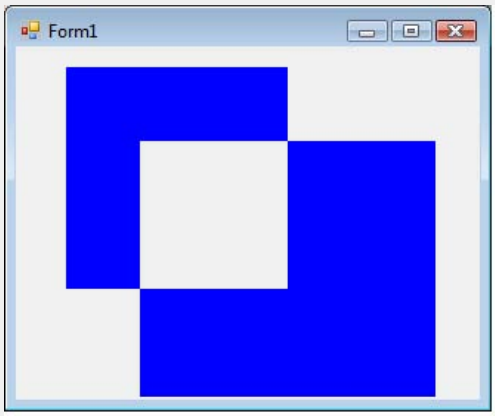
Region2 wird mit Region1 XOR-verknüpft:

```
Reg2.Xor(Reg1);
```

Die Region füllen:

```
g.FillRegion(blueBrush, Reg2);
```

Ergebnis



Clipping

Nach diesem Kurzeinstieg in die Arbeit mit Regionen können wir uns wieder dem Clipping zuwenden. Nutzen Sie Regionen oder Paths, um den Ausgabebereich von Grafikoperationen (Füllen, Zeichnen) in einem *Graphics*-Objekt zu beschränken. Alle über den Clippingbereich hinausgehenden Zeichenoperationen werden abgeschnitten.

BEISPIEL 33.78: Erzeugen eines Clipping-Bereichs, der aus zwei Regionen besteht

C#

```
Graphics g = this.CreateGraphics();
```

Zwei Regionen erzeugen:

```
Region Reg1 = new Region(new Rectangle(100, 100, 200, 200));
Region Reg2 = new Region(new Rectangle(50, 50, 150, 150));
```

Verbinden beider Regionen:

```
Reg2.Union(Reg1);
```

Clipping-Bereich festlegen:

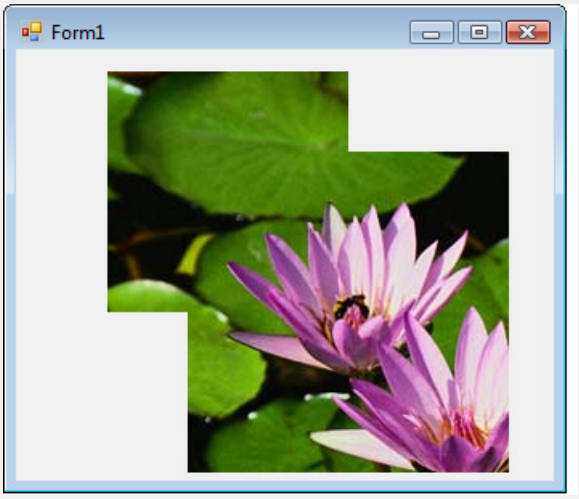
```
g.SetClip(Reg2, CombineMode.Replace);
```

Normales Zeichnen einer Bitmap, die normalerweise die gesamte Fensterfläche füllen würde:

```
g.DrawImage(PictureBox1.Image, 0, 0);
```

Ergebnis

Im Ergebnis wird die Grafikausgabe auf den gewünschten Clipping-Bereich eingeschränkt:

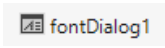


33.6 Standarddialoge

Im Zusammenhang mit der Bearbeitung von Grafiken stehen Ihnen zwei wesentliche Dialoge zur Verfügung:

- Schriftauswahl
- Farbauswahl

33.6.1 Schriftauswahl



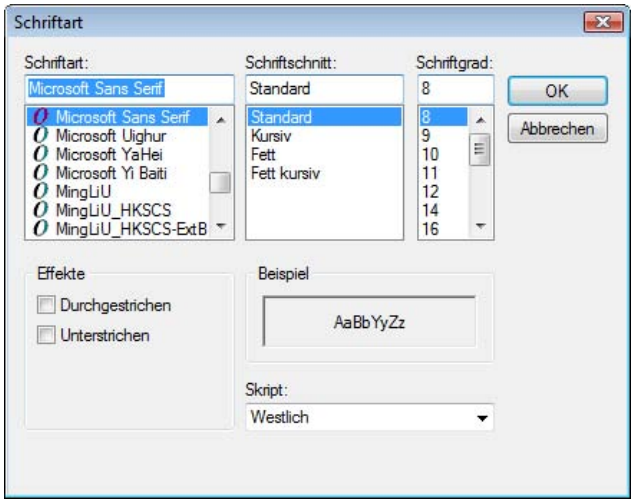
Der wohl jedem bekannte Dialog zur Auswahl einer Schrift bzw. deren Parameter (Größe, Farbe etc.) lässt sich über die entsprechende Komponente *FontDialog* in Ihre Anwendung einbinden.

Die Anzeige erfolgt zur Laufzeit mittels *ShowDialog*-Methode:

```
if (fFontDialog1.ShowDialog() == DialogResult.OK)
 {  ...  }
```

Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln (*DialogResult.OK* oder *DialogResult.Abort*).

Die zweifelsfrei interessanteste Eigenschaft dieser Komponente ist *Font*. Sie können diese Eigenschaft sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten.



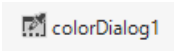
Das Zuweisen der Fontattribute, zum Beispiel an einen Button, gestaltet sich absolut simpel:

```
button1.Font = fontDialog1.Font;
```

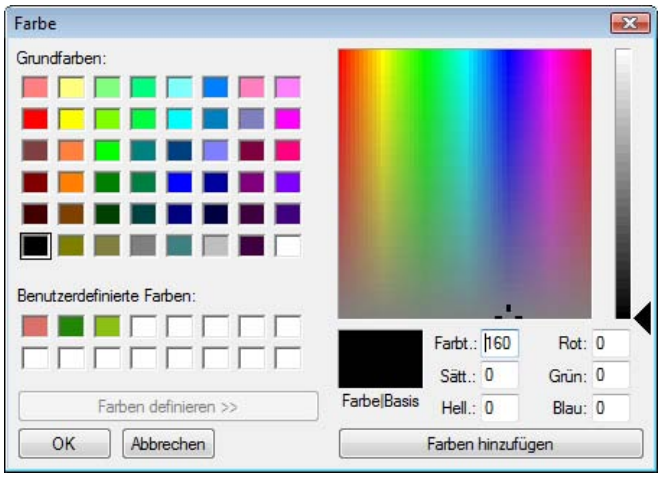
Außer *Font* sind die in der folgenden Tabelle aufgeführten Eigenschaften für das Verhalten bzw. das Aussehen der Dialogbox von Bedeutung:

Eigenschaft	Beschreibung
AllowScriptChange	Skript (Westlich, Arabisch, Türkisch etc.) kann geändert werden.
AllowSimulations	Windows (GDI) darf Schriften simulieren.
AllowVectorFonts	Auswahl von Vektorschriftarten zulassen.
AllowVerticalFonts	Vertikale Fonts sind zulässig.
Color	Die aktuelle Schriftfarbe.
FixedPitchOnly	Es werden nur Schriftarten mit fester Zeichenbreite angezeigt.
FontMustExist	Es können nur vorhandene Schriftarten ausgewählt werden.
MaxSize, MinSize	Maximaler und minimaler Schriftgrad, der ausgewählt werden kann.
ShowColor	Anzeige der Farbauswahl-Combobox
ShowEffects	Anzeige der Effekte (Durchstreichen, Unterstreichen, Farbe)
ShowHelp	Anzeige des Hilfe-Buttons

33.6.2 Farbauswahl



Für die Auswahl von Farben zur Laufzeit können Sie den Standarddialog *ColorDialog* verwenden. Wie auch beim *FontDialog* genügt der einfache Aufruf der Methode *ShowDialog*, um den Dialog anzuzeigen:



Über den Rückgabewert der Methode können Sie den Status beim Beenden der Dialogbox ermitteln (*DialogResult.OK* oder *DialogResult.Abort*).

Im Mittelpunkt der Komponente steht die Eigenschaft *Color* (ein ARGB-Wert), die Sie sowohl vor dem Aufruf des Dialogs initialisieren als auch nach der Anzeige der Dialogbox auswerten können.

BEISPIEL 33.79: Verändern der Formularfarbe

C#
```
colorDialog1.Color = this.BackColor;
if (colorDialog1.ShowDialog() == DialogResult.OK)
    this.BackColor = colorDialog1.Color;
```

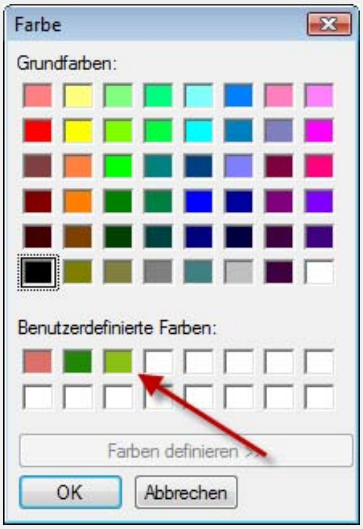
Die wichtigsten Eigenschaften des Dialogs zeigt die folgende Tabelle:

Eigenschaft	Bemerkung
AllowFullOpen, *FullOpen*	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben zulassen
AnyColor	Anzeigen aller verfügbaren Farben
Color	Ruft die von den Benutzern ausgewählte Farbe ab oder legt diese fest
CustomColors	Definieren oder Abfragen von nutzerdefinierten Farben (siehe Beispiel)
FullOpen	Ein-/Ausblenden der rechten Seite des Dialogs zum Definieren eigener Farben beim Öffnen des Dialogs
SolidColorOnly	Nur Volltonfarben können ausgewählt werden

BEISPIEL 33.80: Zuweisen von nutzerdefinierten Farben

C#
```
colorDialog1.AllowFullOpen = false;
colorDialog1.CustomColors = new int[] {6975964, 231202, 1294476};
if (colorDialog1.ShowDialog() == DialogResult.OK) this.BackColor = colorDialog1.Color;
```

Ergebnis



HINWEIS: Wer weitere und detailliertere Informationen über die Grafikausgabe mit GDI+ sucht, der sollte sich das Kapitel 36 zu Gemüte führen.

33.7 Praxisbeispiele

33.7.1 Ein Graphics-Objekt erzeugen

In klassischen Programmiersprachen ist es üblich, mit Methoden direkt auf die Zeichenoberfläche eines Formulars oder eines *Picture*-Controls zuzugreifen. Als .NET-Programmierer müssen Sie umdenken.

Zugriff auf alle wesentlichen Grafik-Methoden erhalten Sie über ein *Graphics*-Objekt. Im Vergleich mit anderen .NET-Objekten hat es allerdings die Besonderheit, dass man es nicht mit dem *new*-Konstruktor erzeugen kann. Woher also nehmen wir es? Das vorliegende Beispiel zeigt Ihnen vier Möglichkeiten:

- Nutzung des im *Paint*-Event des Formulars übergebenen *Graphics*-Objekts,
- Nutzung des in der überschriebenen *OnPaint*-Methode übergebenen *Graphics*-Objekts,
- Erzeugen eines neuen *Graphics*-Objekts mit der *CreateGraphics*-Methode des Formulars,
- Nutzung des im *Paint*-Event einer *PictureBox* (oder einer anderen Komponente, die über dieses Ereignis verfügt) übergebenen *Graphics*-Objekts.

Lassen Sie uns also ein wenig experimentieren!

Zunächst soll uns ein nacktes Windows Form genügen, auf das wir verschiedenfarbige Ellipsen zeichnen wollen. Später ergänzen wir noch weitere Steuerelemente (*Button*, *PictureBox*).

Variante 1: Verwendung des Paint-Events

Die in der *System.Windows.Forms.Form*-Basisklasse implementierte *OnPaint*-Methode wird automatisch nach jedem Freilegen und Verdecken des Fensters aufgerufen, sie löst das *Paint*-Ereignis aus, das wir in einem *Paint*-Eventhandler abfangen und behandeln wollen.

Über das Argument des Events ist ein *Graphics*-Objekt verfügbar:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);      // rote Ellipse
}
```

Test

Die rote Ellipse erscheint sofort nach Programmstart und ist auch nach Freilegen und Verdecken des Fensters zu sehen (siehe folgende Abbildung)

Variante 2: Überschreiben der OnPaint-Methode

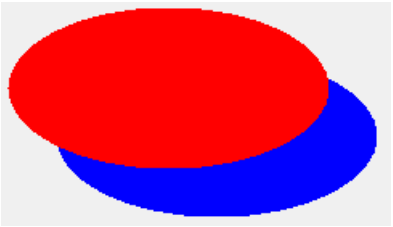
Dies ist die in der .NET Dokumentation favorisierte Realisierung, bei der Sie keinen neuen Eventhandler verwenden müssen, sondern lediglich die *OnPaint*-Methode der Basisklasse überschreiben. Wir wollen nach diesem Prinzip eine versetzte blaue Ellipse zeichnen.

Implementieren Sie die Überschreibung wie folgt:

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.FillEllipse(new SolidBrush(Color.Blue), 40, 60, 200, 100);    // blaue Ellipse
    base.OnPaint(e);         // Aufruf der Basisklassenmethode
}
```

Test

An der Reihenfolge (unten Blau, oben Rot) erkennen Sie, dass die überschriebene *OnPaint*-Methode zuerst abgearbeitet wurde und erst anschließend der bereits vorhandene *Paint*-Eventhandler:



HINWEIS: Wenn Sie die Anweisung *base.OnPaint(e)* auskommentieren, wird das *Paint*-Ereignis nicht mehr ausgelöst, und nur noch die blaue Ellipse erscheint!

Variante 3: Graphics-Objekt mit CreateGraphics erzeugen

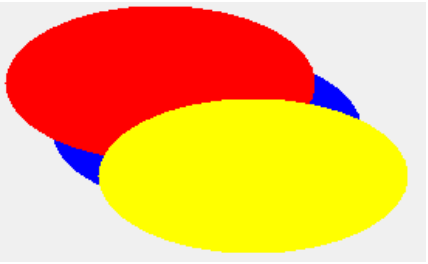
Diese Variante nutzt die Möglichkeit, über die *CreateGraphics*-Methode des Formulars ein neues *Graphics*-Objekt zu erzeugen. Allerdings benötigen wir hier einen *Button*, um das Zeichnen (versetzte gelbe Ellipse) zu demonstrieren.

```
private void button1_Click(object sender, EventArgs e)
{
    Graphics g = this.CreateGraphics();
```

```
        g.FillEllipse(new SolidBrush(Color.Yellow), 70, 90, 200, 100);    // gelbe Ellipse
    }
```

Test

Die gelbe Ellipse erscheint erst nach Klick auf den Button. Im Unterschied zur roten und blauen Ellipse (Variante 1 und 2) verschwindet diese Ellipse wieder, nachdem das Formular vorübergehend verdeckt wurde.



Variante 4: Verwendung des Graphics-Objekts einer PictureBox

Bei einer *PictureBox* – wie bei vielen anderen Komponenten auch – können Sie über die *CreateGraphics*-Methode auf die Zeichenfläche zugreifen. Sinnvoller ist allerdings auch hier die Nutzung des im *Paint*-Event übergebenen *Graphics*-Objekts, da Sie sich dann um die Restaurierung des Bildinhalts nicht weiter zu kümmern brauchen.

Ergänzen Sie die Oberfläche des Testformulars um eine *PictureBox* und erzeugen Sie einen Eventhandler für das *Paint*-Ereignis der *PictureBox*:

```
    private void pictureBox1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.FillEllipse(new SolidBrush(Color.Red), 10, 30, 200, 100);
    }
```

33.7.2 Zeichenoperationen mit der Maus realisieren

Dieses Beispiel zeigt Ihnen eine Möglichkeit, wie Sie einfache Zeichenoperationen (Linie, Ellipse, Rechteck) in ein eigenes Programm integrieren können. Dreh- und Angelpunkt ist die Verwendung einer Hintergrundbitmap, mit deren Hilfe wir einen Gummiband-Effekt beim Zeichnen realisieren.

Oberfläche

Ein Windows Form und eine *ToolStrip*-Komponente zur Auswahl der Zeichenoperation. Zusätzlich fügen Sie noch eine *ColorDialog*-Komponente zur Auswahl der Malfarbe ein.

Wir haben die Oberfläche bewusst einfach gehalten, hier geht es um das Handling der Maus-Events und die Verwendung einer Hintergrundbitmap und nicht um Schönheit im Detail.

Quelltext

```
public partial class Form1 : Form
{
```

Zunächst eine Enumeration definieren:

```
    enum Figuren : int  { Linie, Ellipse, Rechteck }
```

Eine Statusvariable für die Zeichenoperation:

```
    private Figuren Figur = Figuren.Linie;
```

Die Bitmap und das *Graphics*-Objekt für die Hintergrundbitmap

```
    private Bitmap bmp;
    private Graphics bckg;
```

Der Malstift:

```
    private Pen p;
```

Start- und Endpunkt der Zeichenoperation:

```
    private Point p1;
    private Point p2;
```

Im Konstruktor erzeugen wir zunächst die Hintergrundgrafik in der maximal nötigen Größe:

```
    public Form1()
    {
        InitializeComponent();
        Size maxsize = SystemInformation.PrimaryMonitorMaximizedWindowSize;
        bmp = new Bitmap(maxsize.Width, maxsize.Height);
        bckg = Graphics.FromImage(bmp);
```

Mit Hintergrundfarbe füllen:

```
        bckg.Clear(this.BackColor);
```

Zeichenstift initialisieren:

```
        p = new Pen(Color.Black);
    }
```

Mit dem Drücken der Maustaste beginnt der Zeichenvorgang, wir merken uns die Position:

```
    private void Form1_MouseDown(object sender, MouseEventArgs e)
    {
        p1 = e.Location;
    }
```

Jede Mausbewegung bei gedrückter linker Maustaste erfordert das Wiederherstellen der Grafik vor dem Zeichenvorgang und das erneute Zeichnen mit den neuen Mauskoordinaten:

```
    private void Form1_MouseMove(object sender, MouseEventArgs e)
    {
        p2 = e.Location;
```

```
        if (e.Button == MouseButtons.Left)
        {
            Graphics g = CreateGraphics();
            g.DrawImage(bmp, 0, 0);
            Zeichne(g);
            g.Dispose();
        }
    }
```

Erst wenn die Maustaste losgelassen wird, fügen wir das gerade gewählte Zeichenobjekt mit den aktuellen Koordinaten in die Hintergrundbitmap ein:

```
    private void Form1_MouseUp(object sender, MouseEventArgs e)
    {
        Zeichne(bckg);
    }
```

Die eigentliche Zeichenroutine unterscheidet die einzelnen Zeichenobjekte:

```
    private void Zeichne(Graphics dst)
    {
        switch (Figur)
        {
            case Figuren.Linie:
            {
                    dst.DrawLine(p, p1, p2);
                    break;
            }
            case Figuren.Rechteck:
            {
                    dst.DrawRectangle(p, p1.X, p1.Y, p2.X - p1.X, p2.Y - p1.Y);
                    break;
            }
            case Figuren.Ellipse:
            {
                    dst.DrawEllipse(p, p1.X, p1.Y, p2.X - p1.X, p2.Y - p1.Y);
                    break;
            }
        }
    }
```

Auch nach einem Verdecken des Fensters soll die Grafik wieder hergestellt werden:

```
    private void Form1_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        g.DrawImage(bmp, 0, 0);
    }
```

Auswahl der Zeichenobjekte über den *ToolStrip*:

```
    private void toolStripButton1_Click(object sender, EventArgs e)
```

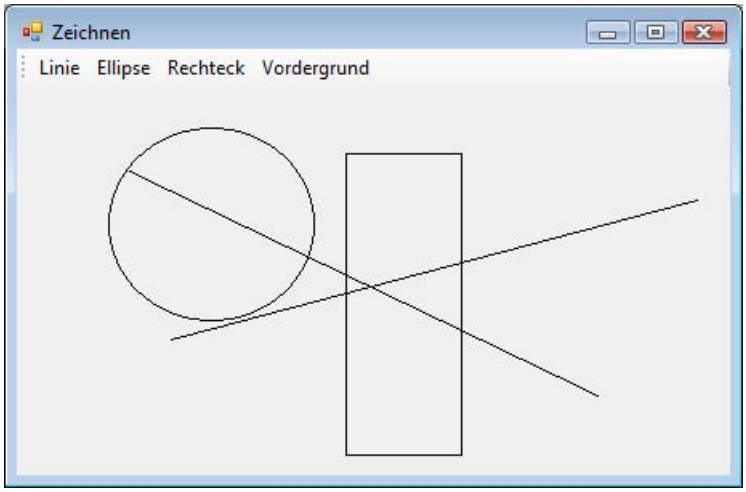
```
    {
        Figur = Figuren.Linie;
    }

    private void toolStripButton3_Click(object sender, EventArgs e)
    {
        Figur = Figuren.Ellipse;
    }

    private void toolStripButton2_Click(object sender, EventArgs e)
    {
        Figur = Figuren.Rechteck;
    }
```

Auswahl der Malfarbe:

```
    private void toolStripButton4_Click(object sender, EventArgs e)
    {
        colorDialog1.AllowFullOpen = true;
        if (colorDialog1.ShowDialog() == DialogResult.OK) p = new Pen(colorDialog1.Color);
    }
}
```

Test

Nach dem Start können Sie Ihren künstlerischen Fähigkeiten freien Lauf lassen:



Kapitel 34

Druckausgabe

In diesem Kapitel wollen wir uns ausgiebig mit den Möglichkeiten beschäftigen, unter C# etwas aufs Papier zu bringen. Vier grundsätzliche Varianten bieten sich an:

- Drucken über die *PrintDocument*-Komponente
- Drucken mit Hilfe von OLE-Automation
- Drucken mit den Crystal Report-Komponenten
- Drucken mit Reporting Services

Hier beschränken wir uns auf die beiden ersten Möglichkeiten. Mehr zur Arbeit mit den Reporting Services finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012].

HINWEIS: Zunächst jedoch sollten Sie sich mit dem Kapitel 33 (Grafikprogrammierung) eingehend beschäftigt haben, da wir auf diesen Grundlagen aufbauen werden.

34.1 Einstieg und Übersicht

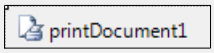
Bevor Sie mit viel Faktenwissen, endlosen Tabellen etc. gepeinigt werden, möchten wir Ihnen an einem Kurzbeispiel das Grundkonzept der Druckausgabe über *PrintDocument* demonstrieren.

34.1.1 Nichts geht über ein Beispiel

BEISPIEL 34.1: Druckausgabe eines 10 x 10 cm großen Rechtecks auf dem Standarddrucker

C#

Fügen Sie zunächst in Ihr Formular eine *PrintDocument*-Komponente ein:



Ergänzen Sie dann das *PrintPage*-Ereignis um folgende Zeilen:

```
private void printDocument1_PrintPage(object sender,
```

BEISPIEL 34.1: Druckausgabe eines 10 x 10 cm großen Rechtecks auf dem Standarddrucker

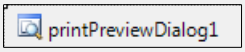
```
                                        System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.PageUnit = GraphicsUnit.Millimeter;
    e.Graphics.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
 }
```

Fügen Sie nun noch einen *Button* ein, mit dem Sie die *Print*-Methode von *PrintDocument1* aufrufen:

```
private void button1_Click(object sender, EventArgs e)
{
     printDocument1.Print();
}
```

Das war es auch schon, nach dem Klick auf den Button dürfte sich Ihr Drucker in Bewegung setzen. Doch was ist der Vorteil einer derartigen ereignisorientierten Programmierung beim Drucken? Die Antwort finden Sie, wenn Sie statt der Druckausgabe zunächst eine Druckvorschau am Bildschirm realisieren möchten.

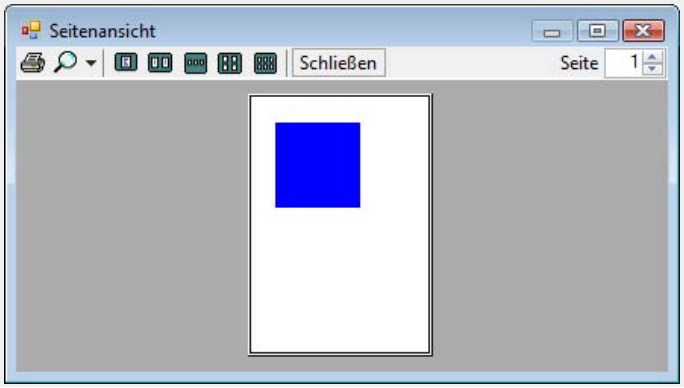
Fügen Sie einfach eine *PrintPreviewDialog*-Komponente in das Formular ein und verknüpfen diese über die Eigenschaft *Document* mit der bereits vorhandenen *PrintDocument*-Komponente.



Der folgende Aufruf zeigt Ihnen bereits die Druckvorschau mit dem Rechteck an:

```
private void button2_Click(object sender, EventArgs e)
{
    printPreviewDialog1.ShowDialog();
}
```

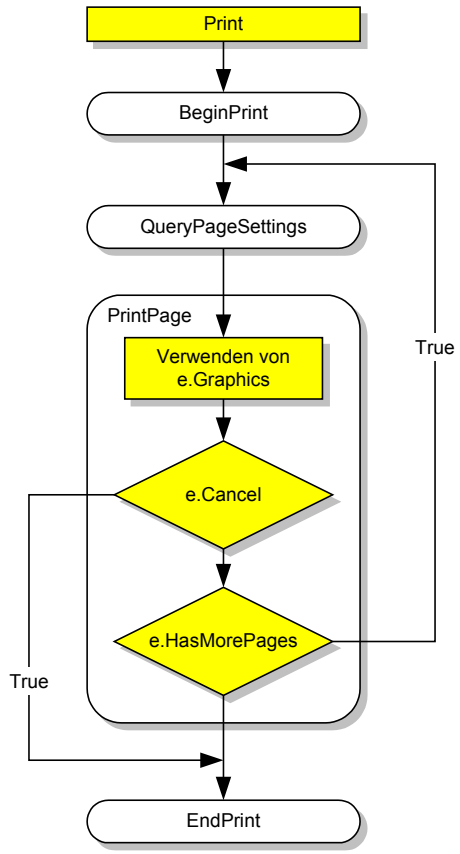
Das erzeugte Druckvorschaufenster:



HINWEIS: Eine Trennung beim Ausgabemedium (Papier, Druckvorschau am Bildschirm) gibt es nicht mehr, Sie entwickeln lediglich **eine** Ausgabelogik im *PrintPage*-Ereignis. Alle Ausgaben erfolgen systemneutral über ein dort bereitgestelltes *Graphics*-Objekt.

34.1.2 Programmiermodell

Wie Sie bereits dem vorhergehenden Beispiel entnehmen konnten, handelt es sich um ein ereignisorientiertes Programmiermodell. Die folgende Skizze soll Ihnen das Grundprinzip noch einmal verdeutlichen:



Mit dem Aufruf der *Print*-Methode wird zunächst das *BeginPrint*-Ereignis von *PrintDocument* ausgelöst. Hier bietet sich Ihnen die Möglichkeit, diverse Einstellungen einmalig zu konfigurieren. Nachfolgend wird das Ereignis *QueryPageSettings* vor dem Druck jeder Seite aufgerufen. Darauf folgt das wohl wichtigste Ereignis: *PrintPage*. Über den Parameter *e* erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin legen Sie hier fest, ob weitere Seiten gedruckt werden sollen (*e.HasMorePages*) oder ob der Druck abgebrochen (*e.Cancel*) werden soll. Steht der Druck weiterer Seiten an, wird die Ereigniskette, wie oben abgebildet, erneut durchlaufen.

Damit wird auch klar, dass Sie selbst dafür verantwortlich sind, welche Seite zu welchem Zeitpunkt gedruckt werden soll. Insbesondere im Zusammenhang mit den Druckersetup-Dialogen werden wir noch einigen Aufwand treiben müssen, aber das sind Sie ja bereits nicht anders gewohnt.

34.1.3 Kurzübersicht der Objekte

Folgende Komponenten stehen Ihnen im Zusammenhang mit der Druckausgabe in Windows Forms-Anwendungen zur Verfügung[1]:

Komponente	Beschreibung
PrintDocument	Der Dreh- und Angelpunkt der Druckausgabe. Über dieses Objekt bestimmen Sie den gewünschten Drucker, die Papierausrichtung, die Auflösung, die zu druckenden Seiten usw. Über das *PrintPage*-Ereignis erhalten Sie Zugriff auf das *Graphics*-Objekt des Druckers. Weiterhin steuern Sie hier den Druckverlauf (Anzahl der Seiten, Seitenauswahl, Abbruch). Mehr zu dieser Komponente finden Sie in den beiden folgenden Abschnitten.
PrintDialog	Der Windows-Standarddialog zur Auswahl eines Druckers sowie der wichtigsten Druckparameter (Seiten, Exemplare, Auflösung)
PageSetupDialog	Der Windows-Standarddialog zur Konfiguration der Druckausgabe (Seitenausrichtung, Papierausrichtung, Seitenränder)
PrintPreviewDialog	Eine komplette Druckvorschau, mit Navigationstasten, Zoom etc.
PrintPreviewControl	Eine Alternative für den *PrintPreviewDialog*. Bei dieser Komponente ist lediglich der Preview-Bereich vorhanden, für die Ansteuerung und Konfiguration sind Sie selbst verantwortlich.

Alle Komponenten können über die *Document*-Eigenschaft mit der *PrintDocument*-Komponente verknüpft werden, Sie müssen also die Parameter nicht "von Hand" übergeben.

34.2 Auswerten der Druckereinstellungen

In den folgenden Abschnitten wollen wir versuchen, Ihnen die "Vorzüge" der relativ unübersichtlichen Objektstruktur zu ersparen. Aus diesem Grund werden wir auf eine Auflistung von Eigenschaften und Methoden für die einzelnen Objekte verzichten, stattdessen stellen wir die zu lösende Aufgabe in den Vordergrund.

34.2.1 Die vorhandenen Drucker

Einen Überblick, welche Drucker auf dem aktuellen System installiert sind, können Sie sich über die Collection *InstalledPrinters* verschaffen.

---

[1] Auf das *Chart*-Control gehen wir im Rahmen des PB 34.6.2 gesondert ein.

BEISPIEL 34.2: Ausgabe aller Druckernamen in einer *ComboBox* und markieren des aktiven Druckers

C#

```
...
using System.Drawing.Printing;
...
    private void Form1_Load(object sender, EventArgs e)
    {
       PrintDocument doc = new PrintDocument();
```

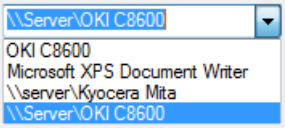
Füllen der *ComboBox*:

```
       foreach (String Printername in PrinterSettings.InstalledPrinters)
           comboBox1.Items.Add(Printername);
```

Auswahl des aktiven Druckers:

```
       comboBox1.Text = doc.PrinterSettings.PrinterName;
    }
```

Ergebnis



34.2.2 Der Standarddrucker

Möchten Sie überprüfen, ob der aktuell gewählte Drucker gleichzeitig auch der System Standarddrucker ist, können Sie dies mit Hilfe der Eigenschaft *IsDefaultPrinter* realisieren.

BEISPIEL 34.3: Test auf Standarddrucker

C#

```
if (printDocument1.PrinterSettings.IsDefaultPrinter)
   MessageBox.Show("Standarddrucker");
```

Ergebnis

Den Standarddrucker erkennen Sie in der Systemsteuerung an einem kleinen Häkchen, auch wenn dieser nicht verfügbar ist:

34.2.3 Verfügbare Papierformate/Seitenabmessungen

Geht es um die Abfrage, welche Papierarten der Drucker unterstützt, können Sie einen Blick auf die *PaperSizes*-Collection werfen. Diese gibt Ihnen nicht nur Auskunft über die Blattgröße (*Height, Width*), sondern auch über die Blattbezeichnung (*PaperName*) und den Typ (*Kind*).

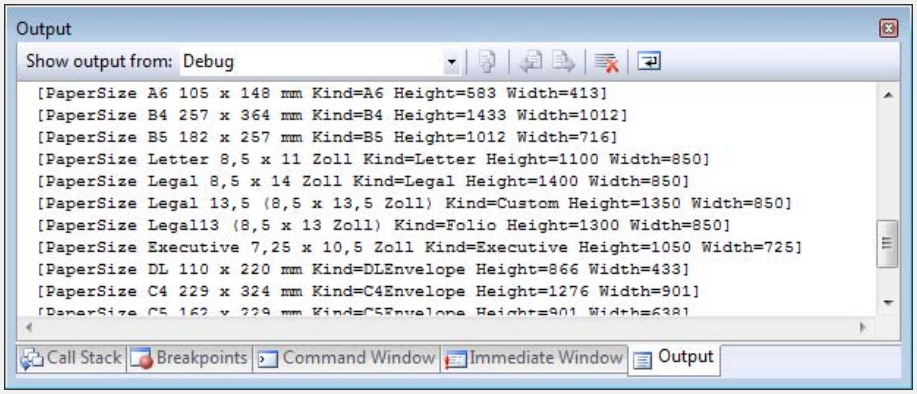
BEISPIEL 34.4: Anzeige aller Papierformate im Ausgabe-Fenster

C#

```
using System.Drawing.Printing;
using System.Diagnostics;
...
    private void button1_Click(object sender, EventArgs e)
    {
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
            Debug.WriteLine(ps);
    }
```

Ergebnis

Die Anzeige im Ausgabe-Fenster:



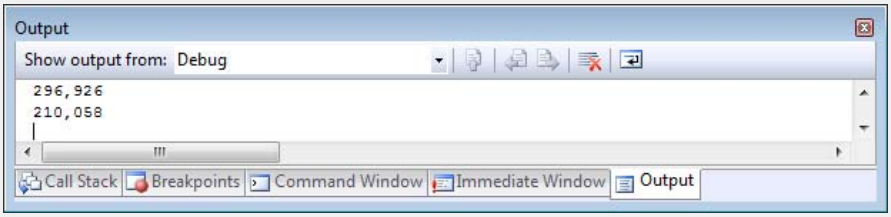
HINWEIS: Die Blattabmessungen werden in 1/100 Zoll zurückgegeben! Der Umrechnungsfaktor in Millimetern ist 0,254.

BEISPIEL 34.5: Anzeige der aktuellen Blattabmessungen in Millimetern

C#

```
Debug.WriteLine(printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Height*0.254);
Debug.WriteLine(printDocument1.PrinterSettings.DefaultPageSettings.PaperSize.Width*0.254);
```

Ergebnis



Gleichzeitig steht Ihnen mit *System.Drawing.Printing.PaperKind* eine Aufzählung der Standardpapierformate zur Verfügung (Auszug):

Element	Beschreibung
A2	A2 (420 x 594 mm)
A3	A3 (297 x 420 mm)
A3Extra	A3 Extra (322 x 445 mm)
A3ExtraTransverse	A3 Extra quer (322 x 445 mm)
A3Rotated	A3 gedreht (420 x 297 mm)
A3Transverse	A3 quer (297 x 420 mm)
A4	A4 (210 x 297 mm)

34.2.4 Der eigentliche Druckbereich

Leider druckt nicht jeder Drucker bis zu den Blatträndern. Aus diesem Grund ist es wichtig, den eigentliche Druckbereich und insbesondere den Offset des Druckbereichs zu bestimmen. Verwenden Sie dazu die Eigenschaften *HardMarginX*, *HardMarginY* sowie *PrintableArea*.

HINWEIS: Vergessen Sie in diesem Zusammenhang die Eigenschaft *Margins* ganz schnell wieder, es handelt sich lediglich um theoretische Seitenränder, die Sie selbst definieren können.

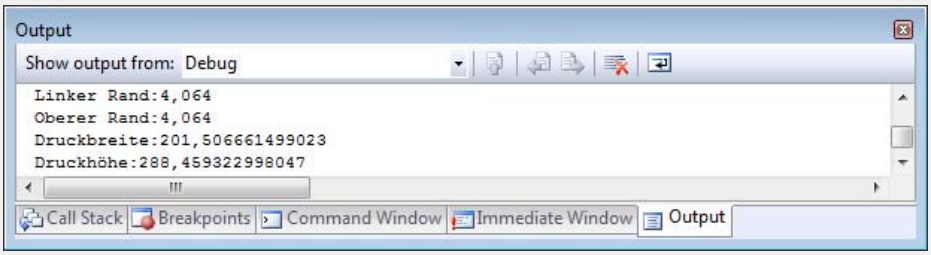
BEISPIEL 34.6: Anzeige der physikalischen Blattränder

C#

```
private void Form1_Load(object sender, EventArgs e)
{
   PrintDocument pd = new PrintDocument();
   Debug.WriteLine("Linker Rand:" + pd.DefaultPageSettings.HardMarginX * 0.254);
   Debug.WriteLine("Oberer Rand:" + pd.DefaultPageSettings.HardMarginY * 0.254);
   Debug.WriteLine("Druckbreite:" + pd.DefaultPageSettings.PrintableArea.Width * 0.254);
   Debug.WriteLine("Druckhöhe:" + pd.DefaultPageSettings.PrintableArea.Height * 0.254);
}
```

Ergebnis

Die Anzeige im Ausgabe-Fenster:



34.2.5 Die Seitenausrichtung ermitteln

Die aktuelle Blatt- bzw. Seitenausrichtung können Sie über die Eigenschaft *Landscape* abfragen.

BEISPIEL 34.7: Abfrage der Seitenausrichtung

C#
```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.DefaultPageSettings.Landscape)
{  ...  };
```

34.2.6 Ermitteln der Farbfähigkeit

Ob Ihr aktuell gewählter Drucker auch in der Lage ist, mehr als nur Schwarz zu Papier zu bringen, lässt sich mit der Eigenschaft *SupportsColor* ermitteln.

BEISPIEL 34.8: Test auf Farbunterstützung

C#
```
using System.Drawing.Printing;
...
if (printDocument1.PrinterSettings.SupportsColor)
{  ...  };
```

34.2.7 Die Druckauflösung abfragen

Möchten Sie sich über die physikalische Druckauflösung des aktiven Druckers informieren, sollten Sie sich mit der Eigenschaft *PrinterResolution* näher beschäftigen.

BEISPIEL 34.9: Die horizontale Druckauflösung (readonly):

C#
```
Debug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.X);
```

BEISPIEL 34.10: Die vertikale Druckauflösung (readonly):

C#
```
Debug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.Y);
```

HINWEIS: Die Rückgabewerte entsprechen Punkten pro Zoll (Dots per Inch: dpi).

Alternativ können Sie über die *Kind*-Eigenschaft einen der folgenden Werte abrufen:

Kind	Beschreibung
Custom	Benutzerdefinierte Auflösung
Draft	Auflösung in Entwurfsqualität
High	Hohe Auflösung

Kind	Beschreibung
Low	Niedrige Auflösung
Medium	Mittlere Auflösung

BEISPIEL 34.11: Ausgabe der Druckauflösung

C#
```
Debug.WriteLine(printDocument1.DefaultPageSettings.PrinterResolution.Kind);
```

34.2.8 Ist beidseitiger Druck möglich?

Ob der Drucker duplexfähig ist, d.h., ob er beidseitig drucken kann, ermitteln Sie über die Eigenschaft *CanDuplex*.

BEISPIEL 34.12: Duplexfähigkeit bestimmen

C#
```
using System.Drawing.Printing;
...
if (PrintDocument1.PrinterSettings.CanDuplex)
{ ... }
```

34.2.9 Einen "Informationsgerätekontext" erzeugen

Wer bisher mit GDI-Funktionen gearbeitet hat, dem wird auch der Begriff "Informationsgerätekontext" nicht unbekannt sein. Der Hintergrund: Bei einem Drucker wird für die Abfrage von Gerätemerkmalen (Auflösung, Seitenränder etc.) häufig ein DC benötigt, der zum Beispiel im Zusammenhang mit der Funktion *GetDeviceCaps* genutzt wird. Dieses DC ist nur für die **Abfrage** von Werten vorgesehen.

Den DC selbst erhalten Sie nur über einen kleinen Umweg: Mit Hilfe der Methode *CreateMeasurementGraphics* erzeugen Sie zunächst ein *Graphics*-Objekt, und dieses stellt bekanntlich die Methode *GetDC* zur Verfügung.

BEISPIEL 34.13: Wir testen die Grafikfähigkeit des aktuellen Druckers[1]

C#
```
using System.Drawing.Printing;
using System.Runtime.InteropServices;

public partial class Form1 : Form
{
```

Die Konstanten für die möglichen Rückgabewerte der API-Funktion:

```
    private const int TECHNOLOGY = 2;
    private const int DT_PLOTTER = 0;
    private const int DT_RASPRINTER = 2;
```

[1] Nicht jeder Drucker muss auch voll grafikfähig sein, hier müssen Sie teilweise Einschränkungen erwarten.

BEISPIEL 34.13: Wir testen die Grafikfähigkeit des aktuellen Druckers

```
    private const int DT_CHARSTREAM = 4;
    private const int DT_METAFILE = 5;
```

Einbinden der API-Funktion:

```
    [DllImport("gdi32.dll")]
    private static extern int GetDeviceCaps(IntPtr hdc, int nIndex);

    private void Form1_Load(object sender, EventArgs e)
    {
```

Informationsgerätekontext erzeugen:

```
        PrintDocument pd = new PrintDocument();
        Graphics g = pd.PrinterSettings.CreateMeasurementGraphics();
        IntPtr dc = g.GetHdc();
```

Abfrage der Grafikfähigkeit und Auswertung:

```
        switch (GetDeviceCaps(dc, TECHNOLOGY))
        {
            case DT_PLOTTER:
                label1.Text = "Plotter"; break;
            case DT_CHARSTREAM:
                label1.Text = "Zeichen"; break;
            case DT_METAFILE:
                    label1.Text = "Metafile"; break;
            case DT_RASPRINTER:
                    label1.Text = "Rasterdrucker"; break;
        }
        g.ReleaseHdc(dc);
    }
}
```

Installieren Sie ruhig einmal den "Generic/Text Only"-Drucker als Standarddrucker und lassen Sie dann das Programm laufen. Dieser Druckertreiber kann nur Zeichen verarbeiten, keine Grafiken.

HINWEIS: Vergessen Sie nicht, den DC wieder freizugeben. Dies muss innerhalb der aktuellen Ereignisroutine geschehen, Sie können den Wert **nicht** in einer globalen Variablen speichern!

34.2.10 Abfragen von Werten während des Drucks

Statt wie in den vorhergehenden Beispielen mit der *PrintDocument*-Komponente, nutzen Sie besser den im *PrintPage*-Ereignis angebotenen Parameter *e*. Über diesen erhalten Sie Zugriff auf die gewünschten Eigenschaften.

BEISPIEL 34.14: Papiergröße während des Drucks bestimmen

C#
```
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   Debug.WriteLine(e.PageSettings.PaperSize);
}
```

34.3 Festlegen von Druckereinstellungen

Nachdem wir im vorhergehenden Abschnitt recht passiv mit den Druckeroptionen umgegangen sind und uns auf das reine Auslesen beschränkt haben, wollen wir uns im Weiteren um das Konfigurieren des Druckers kümmern.

34.3.1 Einen Drucker auswählen

Der wohl erste Schritt, wenn mehr als ein Drucker zur Verfügung steht, ist die Auswahl des Druckers. Zwei Varianten bieten sich an:

- Verwendung der Eigenschaft *PrinterName*
- Verwendung der *PrintDialog*-Komponente (siehe Abschnitt 34.4)

HINWEIS: Nach dem Setzen der Eigenschaft bzw. vor dem endgültigen Drucken sollten Sie mit der Eigenschaft *IsValid* überprüfen, ob die Konfiguration auch realisierbar ist.

BEISPIEL 34.15: Setzen der *PrinterName*-Eigenschaft und nachfolgende Prüfung mit *IsValid*

C#
```
printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
if (printDocument1.PrinterSettings.IsValid)
  printPreviewDialog1.ShowDialog();
```

BEISPIEL 34.16: Verwendung des *QueryPageSettings*-Ereignisses zur Auswahl eines Druckers

C#
```
private void printDocument1_QueryPageSettings(object sender, QueryPageSettingsEventArgs e)
 {
      e.PageSettings.PrinterSettings.PrinterName = "FRITZFax Drucker";
 }
```

34.3.2 Drucken in Millimetern

Sie werden hoffentlich nicht auf die Idee kommen, Zeichnungen in Pixeln auf dem Drucker auszugeben, je nach Modell ist sonst Ihre Grafik mikroskopisch klein oder riesengroß. Bleibt die Frage, wie Sie die Maßeinheit auf Millimeter umstellen können. Die Lösung ist schnell gefunden, über die Eigenschaft *PageUnit* können Sie eine der folgenden Maßeinheiten auswählen:

Konstante	Beschreibung
Display	Eine Einheit entspricht 1/75 Zoll.
Document	Eine Einheit entspricht 1/300 Zoll.
Inch	Eine Einheit entspricht 1 Zoll.
Millimeter	Eine Einheit entspricht einem Millimeter.
Pixel	Eine Einheit entspricht einem Gerätepixel.
Point	Eine Einheit entspricht 1/72 Zoll (Point).

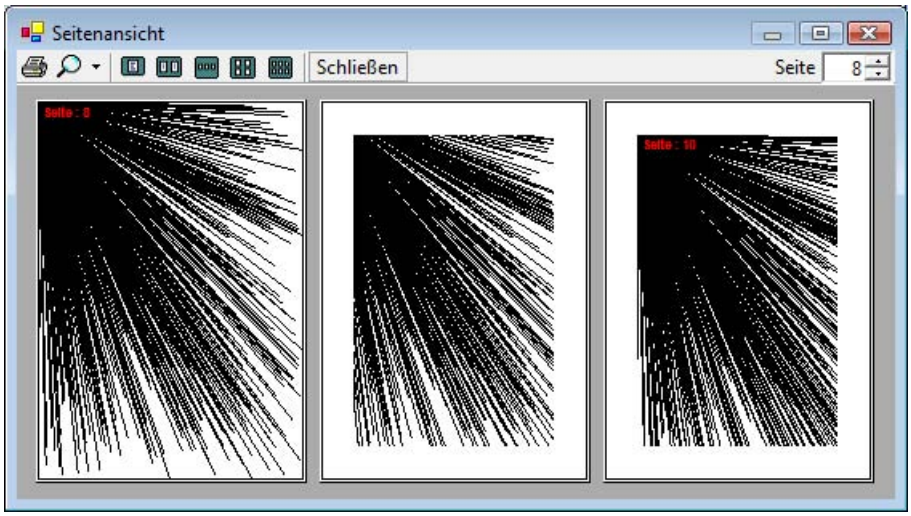
BEISPIEL 34.17: Setzen der Maßeinheit im *PrintPage*-Ereignis

C#
```
private void printDocument1_PrintPage(object sender, PrintPageEventArgs e)
{
   e.Graphics.PageUnit = GraphicsUnit.Millimeter;
   e.Graphics.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
 ...
}
```

34.3.3 Festlegen der Seitenränder

Tja, welche Ränder meinen Sie denn? Geht es um Seitenränder wie zum Beispiel in Microsoft Word, nutzen Sie die Eigenschaft *Margins*. Allerdings bedeutet das Festlegen per Code oder mit Hilfe der Dialogbox *PageSetupDialog* noch lange nicht, dass diese Ränder auch zwingend eingehalten werden. Dafür sind Sie im *PrintPage*-Ereignis selbst verantwortlich.

Die folgende Abbildung soll Ihnen die Problematik verdeutlichen. In jedem der drei Fälle werden, beginnend mit der Koordinate 0,0 (linke obere Ecke), Zufallslinien gezeichnet, die maximal die Abmessungen des Blattes erreichen.



BEISPIEL 34.18: Variante 1

C# Variante 1 zeigt deutlich, dass die eingestellten Seitenränder (100,100,100,100) vollkommen ignoriert werden:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
for (i = 0; i <= 500; i++)
     g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                         Rnd.Next(e.PageBounds.Height));
```

BEISPIEL 34.19: Variante 2

C# Diese Variante berücksichtigt bereits die eingestellten Seitenränder durch die Verwendung eines Clipping-Bereichs:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
g.SetClip(e.MarginBounds);
for (i = 0; i <=500; i++)
      g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                 Rnd.Next(e.PageBounds.Height));
```

BEISPIEL 34.20: Variante 3

C# Wir bringen auch den Koordinatenursprung an die richtige Position:

```
Graphics g = e.Graphics;
g.PageUnit = GraphicsUnit.Display;
g.SetClip(e.MarginBounds);
g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
for (i = 0; i <= 500; i++)
       g.DrawLine(p, 0, 0, Rnd.Next(e.PageBounds.Width),
                  Rnd.Next(e.PageBounds.Height));
```

Damit brauchen Sie sich beim Zeichnen eigentlich nur noch um die Breite und Höhe des bedruckbaren Bereichs (*e.MarginBounds.Width* bzw. *e.MarginBounds.Height*) zu kümmern, die linke obere Ecke ist bereits korrekt gesetzt.

HINWEIS: Die Eigenschaft *Margins* hebt natürlich keine physikalischen Grenzen auf. Wenn der Drucker einen entsprechenden Offset aufweist, müssen Sie diesen auch berücksichtigen (siehe Abschnitt 34.2.4).

34.3.4 Druckjobname

Was im Normalfall eher sekundär ist, kann in Netzwerkumgebungen bzw. Multiuser-Umgebungen für mehr Übersicht sorgen. Über die Eigenschaft *DocumentName* können Sie vor dem Drucken einen aussagekräftigen Druckjobnamen festlegen, der im Druckerspooler angezeigt wird.

BEISPIEL 34.21: Ändern des Druckjob-Namens

```
C# printDocument1.DocumentName = "Mein erster C#.NET-Druckversuch";
```

34.3.5 Anzahl der Kopien

Die Anzahl der Druckkopien kann zum einen mit Hilfe des Dialogs *PrintDialog*, zum anderen auch per Code festgelegt werden. Nutzen Sie die Eigenschaft *Copies*.

BEISPIEL 34.22: Drei Kopien

```
C# printDocument1.PrinterSettings.Copies = 3;
```

HINWEIS: Mit *MaximumCopies* können Sie einen Maximalwert für die Druckdialoge vorgeben!

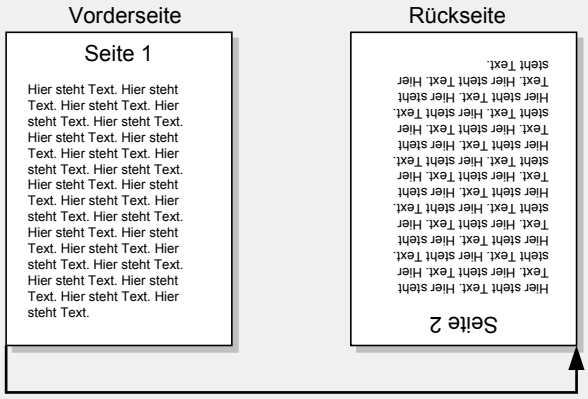
BEISPIEL 34.23: Maximal fünf Kopien zulassen

```
C# printDocument1.PrinterSettings.MaximumCopies = 5;
```

34.3.6 Beidseitiger Druck

Geht es um das beidseitige Bedrucken von Papier, was aus ökologischer Sicht sicher sinnvoller ist, müssen Sie sich zunächst vergewissern, ob der Drucker auch über dieses Feature verfügt (siehe Abschnitt 34.2.8). Nachfolgend können Sie über die *Duplex*-Eigenschaft den gewünschten Wert einstellen.

Konstante	Beschreibung
Default	Die Standardeinstellungen des Druckers werden genutzt.
Simplex	Der "normale" einseitige Druck.
Horizontal	

Konstante	Beschreibung
Vertical	

Vorderseite
Seite 1
Hier steht Text. Hier steht Text. Hier steht Text.

Ein Beispiel zeigt die Auswertung der vier Varianten im Zusammenhang.

BEISPIEL 34.25: Berücksichtigung des vorgegebenen Druckbereichs

C#

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine Variable für die aktuell zu druckende Seite:

```
    int page;
```

Die aktuelle Seite bei Mehrfachauswahl:

```
    int selectedindex;
```

Die maximal druckbaren Seiten (Dokumentlänge):

```
    const int maxpages = 30;
```

Beim Programmstart füllen wir zunächst eine *ListBox* mit den möglichen Seitenzahlen (1...30):

```
    private void Form1_Load(object sender, EventArgs e)
    {
        for (int i = 0; i <= maxpages; i++)
                listBox1.Items.Add("Seite " + i.ToString());
    }
```

Druckerdialog anzeigen und im Erfolgsfall die Druckvorschau öffnen:

```
    private void button1_Click(object sender, EventArgs e)
    {
        if (printDialog1.ShowDialog() == DialogResult.OK)
                printPreviewDialog1.ShowDialog();
    }
```

Vorbereiten des "Druckvorgangs":

```
    private void printDocument1_BeginPrint(object sender,
                                           System.Drawing.Printing.PrintEventArgs e)
    {
        page = 1;
        selectedindex = 0;
```

Zur Sicherheit prüfen wir, ob auch mindestens eine Seiten ausgewählt wurde (nur bei Seitenauswahl):

```
        switch (printDialog1.PrinterSettings.PrintRange)
        {
            case PrintRange.Selection:
                    if (listBox1.SelectedItems.Count == 0)
                        e.Cancel = true;
                break;
```

BEISPIEL 34.25: Berücksichtigung des vorgegebenen Druckbereichs

```
        }
    }
```

Der eigentliche Druckvorgang:

```
    private void printDocument1_PrintPage(object sender,
                                          System.Drawing.Printing.PrintPageEventArgs e)
    {
        int printpage = 0;
```

Ja nach Auswahl im Druckdialog müssen wir nun die aktuelle Seite bestimmen:

```
        switch (e.PageSettings.PrinterSettings.PrintRange)
        {
```

Es soll die aktuelle Seite gedruckt werden (der Wert wird per *NumericUpDown* bestimmt):

```
            case PrintRange.CurrentPage:
                    printpage = (int) numericUpDown1.Value;
                    break;
```

Es soll ein Seitenbereich gedruckt werden:

```
            case PrintRange.SomePages:
                    printpage = page + e.PageSettings.PrinterSettings.FromPage - 1;
                    break;
```

Es sollen alle Seiten gedruckt werden:

```
            case PrintRange.AllPages:
                    printpage = page;
                    break;
```

Eine Seitenauswahl (*ListBox*) soll gedruckt werden:

```
            case PrintRange.Selection:
                    printpage = listBox1.SelectedIndices[selectedindex];
                    selectedindex++;
                    break;
        }
```

Hier können Sie die Seite auswerten und die Drucklogik unterbringen:

```
        switch (printpage)
        {
            case 1:
                    break;
            case 2:
                    break;
            // ...
        }
```

BEISPIEL 34.25: Berücksichtigung des vorgegebenen Druckbereichs

Unser Beispiel zeigt stattdessen die aktuelle Seitenzahl an:

```
e.Graphics.DrawString("Seite : " + printpage.ToString(),
       new Font("Arial", 20, FontStyle.Bold, GraphicsUnit.Millimeter),
                Brushes.Black, 70, 50);
```

Eine Seite weiter:

```
page++;
```

Ja nach Auswahl im Druckerdialog bestimmen wir jetzt, ob es noch weitere Seiten gibt:

```
switch (e.PageSettings.PrinterSettings.PrintRange)
{
    case PrintRange.Selection:
           e.HasMorePages = selectedindex < listBox1.SelectedIndices.Count;
           break;
    case PrintRange.CurrentPage:
           e.HasMorePages = false;
           break;
    case PrintRange.SomePages:
           e.HasMorePages = (printpage < e.PageSettings.PrinterSettings.ToPage);
           break;
    case PrintRange.AllPages:
           e.HasMorePages = (page <= maxpages);
           break;
}
  }
}
```

HINWEIS: Über die Eigenschaften *MinimumPage* und *MaximumPage* können Sie maximale Grenzen für die Auswahl des Druckbereichs festlegen.

BEISPIEL 34.26: Druckbereich maximal von Seite 1 bis Seite 10

C#
```
printDocument1.PrinterSettings.MinimumPage = 1;
printDocument1.PrinterSettings.MaximumPage = 10;
```

34.3.8 Druckqualität verändern

Unter diesem Punkt verstehen wir zum einen die Einstellung der dpi-Zahl des Druckers, zum anderen die Optionen bei der Ausgabe von Grafiken (*Antialiasing, CompositingQuality*).

BEISPIEL 34.27: Setzen der Druckauflösung

C#
```
printDocument1.DefaultPageSettings.PrinterResolution =
          printDocument1.PrinterSettings.PrinterResolutions[2];
```

34.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen

An dieser Stelle wollen wir uns einem Spezialfall zuwenden. Im Mittelpunkt steht die *Chart*-Komponente und deren Möglichkeiten zur Druckausgabe. Diese sind zwar auf den ersten Blick recht überschaubar, allerdings dürften Sie damit auch alle wichtigen Anwendungsfälle problemlos abdecken. Unser Interesse gilt vor allem der *Printing*-Eigenschaft des *Chart*-Controls, diese bündelt alle Aktivitäten rund um die Druckausgabe.

Folgende Methoden werden bereitgestellt:

Methode	Beschreibung
PageSetup	... zeigt den bekannten Page Setup-Dialog an. Mehr zur Verwendung dieses Dialogs finden Sie im Abschnitt 34.4.2 ab Seite 1699.
Print	... druckt das vorliegende Diagramm. Übergeben Sie als Parameter *true*, wird der bekannte Druckdialog zur Druckerauswahl angezeigt (siehe Abschnitt 34.4.1 Seite 1698).
PrintPaint	Ausgabe des Diagramms auf einem *Graphics*-Objekt (siehe PB 34.6.2).
PrintPreview	Statt der direkten Druckauswahl wird eine Druckvorschau angezeigt.

Neben obigen Methoden steht über die *Printing*-Eigenschaft auch ein *PrintDocument* zur Verfügung, über das Sie die Einstellungen des Druckers auslesen oder auch beeinflussen können, wie Sie es in den beiden vorhergehenden Abschnitten bereits im Detail gesehen haben.

BEISPIEL 34.28: Verwendung von *Printing.PrintDocument*

C#

Anzeige, ob es sich um den Standard-Drucker handelt:

```
Text = chart1.Printing.PrintDocument.PrinterSettings.IsDefaultPrinter.ToString();
```

HINWEIS: Ein Praxisbeispiel für die Integration eines Diagramms in einen eigenen Report finden Sie unter PB 34.6.2 Diagramme mit dem Chart-Control drucken.

HINWEIS: Ein umfangreiches Kapitel zum *Chart*-Control finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012].

34.4 Die Druckdialoge verwenden

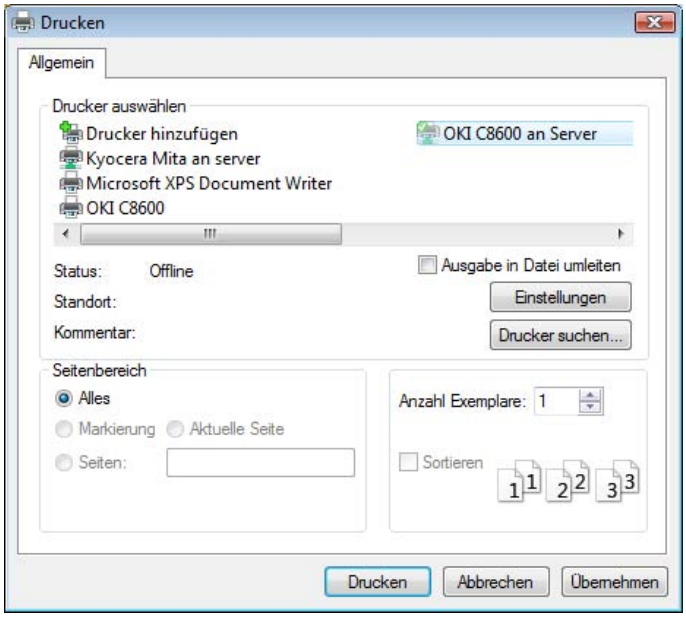
Im Folgenden wollen wir Ihnen kurz die drei wesentlichen Druckdialoge

- *PrintDialog*
- *PageSetupDialog*
- *PrintPreviewDialog*

und deren wichtigste Parameter im Zusammenspiel mit der Druckausgabe vorstellen.

34.4.1 PrintDialog

Der allgemein bekannte Standarddruckdialog wird mit der Komponente *PrintDialog* eingebunden.



Die Komponente selbst können Sie mittels *Document*-Eigenschaft direkt an ein *PrintDocument*-Control binden. Alle gewählten Parameter werden automatisch an *PrintDocument* übergeben.

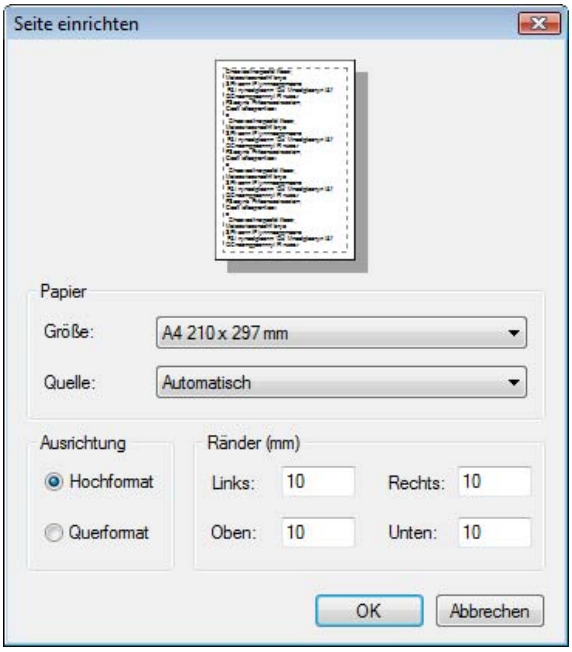
Eigenschaft	Beschreibung
AllowPrintToFile	... aktiviert das Kontrollkästchen "Ausgabe in Datei".
AllowSelection	... aktiviert das Optionsfeld "Seiten von ... bis ...".
AllowSomePages	... aktiviert das Optionsfeld "Seiten".
ShowHelp	... aktiviert die Schaltfläche "Hilfe".
ShowNetwork	... aktiviert die Schaltfläche "Netzwerk" (nur in der Theorie).
PrinterSettings	Über diese Eigenschaft können Sie Standardwerte vorgeben sowie die Einstellungen des Dialogfelds abfragen.
PrintToFile	... fragt den Wert des Kontrollkästchens "Ausgabe in Datei" ab.

BEISPIEL 34.29: Anzeige des Dialogs und Abfrage des gewählten Druckers

C#
```
if (printDialog1.ShowDialog() = DialogResult.OK)
    MessageBox.Show(printDialog1.PrinterSettings.PrinterName, "Hinweis",
                    MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```

34.4.2 PageSetupDialog

Aus vielen Programmen dürfte Ihnen der folgende Dialog bekannt sein, mit dem Sie einen Menüpunkt "Seite einrichten" realisieren können.



Auch diese *PageSetupDialog*-Komponente können Sie mittels *Document*-Eigenschaft direkt an eine *PrintDocument*-Komponente binden, um die eingestellten Parameter automatisch zu übernehmen.

Eigenschaft	Beschreibung
AllowMargins	... aktiviert den Bereich "Ränder (mm)".
AllowOrientation	... aktiviert den Bereich "Orientierung".
AllowPaper	... aktiviert den Bereich "Papier".
AllowPrinter	... aktiviert die Schaltfläche "Drucker...".
ShowHelp	... aktiviert die Schaltfläche "Hilfe".
MinMargins	... legt die minimalen Werte für die Ränder fest.
PageSettings *PrinterSettings*	... hier können Sie Standardwerte vorgeben bzw. die Werte abfragen.

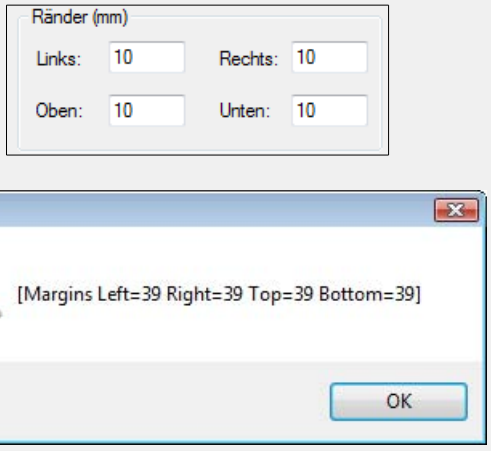
HINWEIS: Über das Ereignis *HelpRequest* können Sie auf den Button "Hilfe" reagieren!

BEISPIEL 34.30: Aufruf der Dialogbox und Anzeige der neu gesetzten Ränder

C#

```
if (pageSetupDialog1.ShowDialog()== DialogResult.OK)
      MessageBox.Show(pageSetupDialog1.PageSettings.Margins.ToString(), "Hinweis",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
```

Ergebnis



Probleme mit den Rändern

Doch wo viel Licht, da ist auch Schatten, ein kleiner Bug hat sich in die Komponente eingeschlichen, der aber wahrscheinlich nur in den lokalisierten Varianten von Visual Studio auftritt:

HINWEIS: Die Werte der eingestellten Ränder stimmen nicht mit den Werten der Eigenschaft *Margins* überein (aus einem Zoll Vorgabewert werden in der Anzeige 10 Millimeter und aus diesen wiederum korrekte 0,39 Zoll). Eine fragwürdige Umrechnung.

Deshalb der folgende Workaround:

Rufen Sie **vor** dem Aufruf der Dialogbox jedes Mal die folgenden Anweisungen auf:

```
pageSetupDialog1.PageSettings.Margins.Left =
   (int) (pageSetupDialog1.PageSettings.Margins.Left * 2.54);
pageSetupDialog1.PageSettings.Margins.Top =
   (int) (pageSetupDialog1.PageSettings.Margins.Top * 2.54);
pageSetupDialog1.PageSettings.Margins.Right =
   (int) (pageSetupDialog1.PageSettings.Margins.Right * 2.54);
pageSetupDialog1.PageSettings.Margins.Bottom =
    (int) (pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
pageSetupDialog1.ShowDialog();
```

Nach dem Aufruf stehen Ihnen die Seitenränder wieder in der korrekten 1/100-Zoll-Angabe zur Verfügung.

34.4.3 PrintPreviewDialog

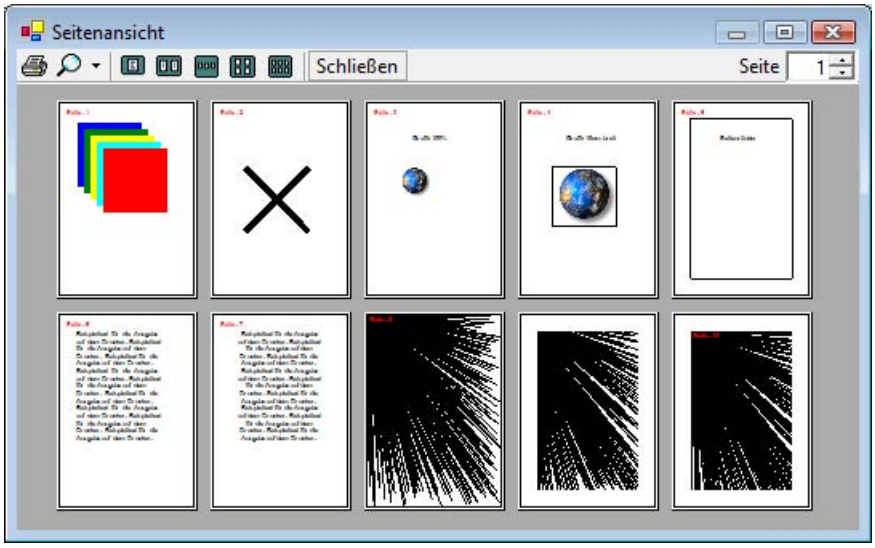
Im Grunde ist die Verwendung des *PrintPreviewDialog* recht simpel. Nach dem Einfügen der Komponente brauchen Sie diese lediglich über die *Documents*-Eigenschaft mit der *PrintDocument*-Komponente zu verknüpfen und die Methode *ShowDialog* aufrufen.

Bis auf die Eigenschaft *UseAntialias* (Verbessern der Anzeigequalität) können Sie kaum weitere Einstellungen vornehmen. Die Ausnahme stellt die Eigenschaft *PrintPreviewControl* dar, mit der Sie direkt das Aussehen und Verhalten der Vorschau beeinflussen können.

BEISPIEL 34.31: Gleichzeitige Anzeige von zehn Seiten

C#
```
printPreviewDialog1.PrintPreviewControl.Rows = 2;
printPreviewDialog1.PrintPreviewControl.Columns = 5;
printPreviewDialog1.ShowDialog();
```

HINWEIS: Mehr zur Konfiguration und Verwendung der *PrintPreviewControl*-Komponente finden Sie im folgenden Abschnitt 34.4.4.



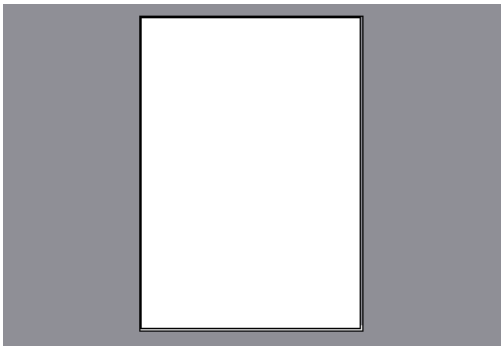
Weiterhin dürfte die Eigenschaft *WindowState* in Zusammenhang mit der Anzeige der Dialogbox von Interesse sein. Hiermit steuern Sie die Art der Anzeige (Vollbild etc.).

BEISPIEL 34.32: Vollbildanzeige aktivieren

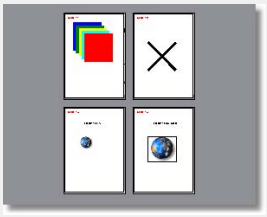
C#
```
printPreviewDialog1.WindowState = FormWindowState.Maximized;
printPreviewDialog1.ShowDialog();
```

34.4.4 Ein eigenes Druckvorschau-Fenster realisieren

Wem die *PrintPreview*-Komponente zu wenig Eingriffsmöglichkeiten bietet, der kann sich mit der *PrintPreviewControl*-Komponente eine eigene Druckvorschau zusammenbauen.



Bis auf den reinen Druckvorschaubereich können Sie sich um alle Einstellungen und optischen Spielereien selbst kümmern. Die folgende Tabelle listet die wichtigsten Eigenschaften auf:

Eigenschaft	Beschreibung
AutoZoom	Ist der Wert auf *True* gesetzt, werden die Seiten so skaliert, dass die vorgegebene Anzahl von Seiten flächenfüllend dargestellt wird. *AutoZoom = False* *AutoZoom = True* 
BackColor	... die Hintergrundfarbe für die Druckvorschau.
Columns	... die Anzahl von Spalten, d.h., wie viele Seiten nebeneinander dargestellt werden.
Rows	... die Anzahl von Zeilen, d.h., wie viele Seiten untereinander dargestellt werden.
Document	... die Verknüpfung zum *PrintDocument*-Objekt.
StartPage	... die Seitenzahl der linken oberen Seite. Durch Verändern dieses Wertes können Sie die weiteren Seiten anzeigen.
Zoom	... legt explizit einen Zoomfaktor fest.

Wie Sie die *PrintPreviewControl*-Komponente im Zusammenhang verwenden, zeigt Ihnen das Praxisbeispiel 34.6.1 am Kapitelende.

34.5 Drucken mit OLE-Automation

Wir wollen versuchen, mit Hilfe der bekannten Office-Programme Druckausgaben zu realisieren. Schnell kommt der Verdacht auf, dass der Programmierer versucht, das Brett an der dünnsten Stelle anbohren zu wollen. Doch warum sollen nicht die Möglichkeiten von Office-Programmen genutzt werden, wenn doch häufig der Wunsch besteht, Reportausgaben nachträglich zu bearbeiten oder in umfangreichere Dokumentationen aufzunehmen? Nicht zuletzt bieten sich gerade die Office-Anwendungen an, wenn es um eine sinnvolle Archivierung von Dokumenten geht.

Unser Favorit ist, wie sollte es auch anders sein, Microsoft Word, ein Allround-Talent, was die Gestaltung von ansprechenden Druckausgaben angeht. Als zweite Variante bietet sich insbesondere bei der Verwendung von Desktop-Datenbanken Microsoft Access an. Hier kann der integrierte Report-Generator zeigen, was er kann.

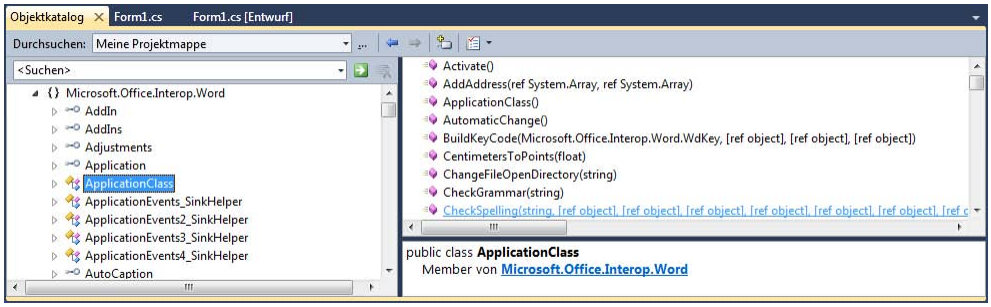
Für die im Weiteren vorgestellten Verfahren ist es sinnvoll, wenn wir kurz auf die grundlegenden Möglichkeiten und Funktionen der OLE-Automation eingehen.

HINWEIS: Ein wesentlicher Nachteil der im Folgenden beschriebenen Verfahrensweise soll natürlich nicht unerwähnt bleiben. Geben Sie Ihre Anwendungen an andere Anwender weiter, muss auf dem jeweiligen Rechner natürlich auch die OLE-Anwendung (Access oder Word) installiert sein.

34.5.1 Kurzeinstieg in die OLE-Automation

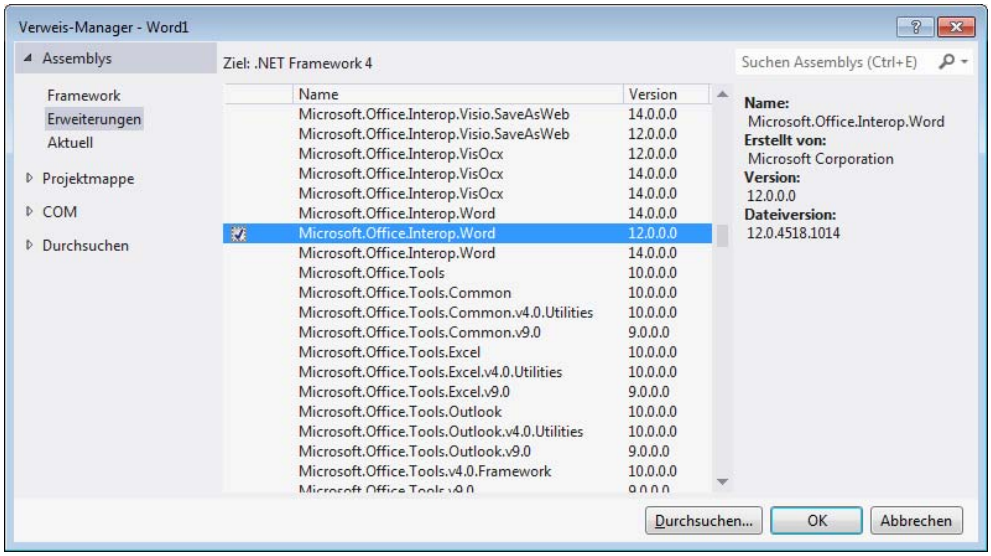
Über OLE-Automation lassen sich Objekte anderer Applikationen (z.B. Word oder Access) von Ihrem .NET-Programm quasi "fernsteuern". Nach der Definition einer entsprechenden Objektvariablen können Sie auf Eigenschaften und Methoden dieser Objekte genauso zugreifen, als ob es sich um ein normales C#-Objekt handeln würde.

Wichtigstes Hilfsmittel für den OLE-Programmierer ist der in Visual Studio integrierte Objektkatalog (siehe folgende Abbildung).



Im Objektkatalog werden neben den Klassen alle Methoden, Eigenschaften, Ereignisse und Konstanten des jeweiligen Objekts angezeigt.

Welche Objekte angezeigt werden, hängt von den Verweisen ab, die Sie unter *Projekt/Verweise hinzufügen...* eingebunden haben:



Programmieren der OLE-Automation

Das Grundprinzip besteht darin, dass Sie in C# eine Instanz der gewünschten Klasse erzeugen. Mit diesem Objekt können Sie dann wie mit jedem anderen Objekt arbeiten.

HINWEIS: Wer bereits mit älteren C#-Versionen (< 4.0) gearbeitet hat, wird sicher sicher noch an die grauenhafte Arbeit mit dem Erzeugen von Objekten und der Übergabe von optionalen Parametern erinnern. Vergessen Sie dies alles, seit C# 4.0, können Sie auf derartige Handstände verzichten

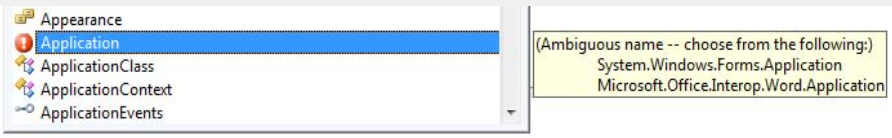
Voraussetzung für das Erstellen einer Instanz ist ein Verweis auf die entsprechende Klasse. Um neue Verweise zu erstellen, müssen Sie unter *Projekt/Verweise hinzufügen...*

- die gewünschte COM-Klassenbibliothek (z.B. *Microsoft Word 12.0 Object Library*) auswählen
- oder alternativ eine der vorhandenen PIAs (*Primary Interop Assemblies*) nutzen.

Im vorliegenden Fall nutzen wir die PIA *Microsoft.Office.Interop.Word* (siehe obige Abbildung),

HINWEIS: Binden Sie den neuen Namenspace (*using*) direkt ein, kann es zu Überschneidungen der COM-Objektnamen mit den C#-Objekten kommen.

BEISPIEL 34.33: Hier ist nicht eindeutig welches *Application* gemeint ist



Besser Sie verwenden in diesem Fall einen Aliasnamen für die Assembly, wie es das folgende Beispiel zeigt:

BEISPIEL 34.34: Verwendung Aliasnamen

C# Erstellen einer Objektvariablen *myWord* als Instanz des *Word.Application*-Objekts:

```
using Word = Microsoft.Office.Interop.Word;

namespace WindowsFormsApplication3
{
...
     var myWord = new Word.Application();
...
```

34.5.2 Drucken mit Microsoft Word

Verwenden Sie Word für die Druckausgabe, können Sie zwei verschiedene Varianten einsetzen:

- Sie entwerfen die komplette Seite mit Word und fügen an den relevanten Stellen so genannte Platzhalter (Formularfelder) ein. Diese werden später aus dem C#-Programm heraus gezielt aufgerufen und mit neuen Inhalten gefüllt. Der Vorteil dieser Variante: das Word-Dokument kann quasi wie eine Vorlage genutzt werden, der Aufwand ist minimal. Nachteil: Sie müssen die Datei zur Laufzeit in den Word-Editor laden.
- Der komplette Report bzw. das komplette Word-Dokument wird zur Laufzeit aus C# heraus generiert. Vorteil: das Erstellen von Listen ist mit dieser Variante wesentlich einfacher als mit vorgefertigten Dokumenten. Nachteil: jede Menge Quellcode.

Der Nachteil der zweiten Variante kann jedoch schnell wieder wettgemacht werden, lassen Sie einfach Word für sich arbeiten.

Was ist gemeint? Die Antwort findet sich unter dem Word-Menüpunkt *Extras|Makro|Aufzeichnen*[1]. Öffnen Sie Word und rufen Sie den genannten Menübefehl auf. Alle weiteren Aktionen, die Sie durchführen (Text eingeben, formatieren, Tabellen erstellen etc.) werden durch den Makrorekorder aufgezeichnet. Sie müssen nur noch das aufgezeichnete Makro in Ihr C#-Programm einfügen und "geringfügig" anpassen.

[1] Oder für Opfer der Versionen ab 2007: *Entwicklertools|Makro aufzeichnen* (Eventuell müssen Sie die Registerkarte *Entwicklertools* zunächst über die Optionen einblenden).

BEISPIEL 34.35: Transformieren eines aufgezeichneten Word-Markros in C#

C#

Sie erstellen bei eingeschaltetem Makrorecorder ein neues Dokument und geben eine 16 Punkt große Überschrift ein. Das Resultat ist folgender Bandwurm (Word-VBA):

```
Sub Makro1()
  Documents.Add Template:="E:\Programme\Ms\Vorlagen\Normal.dot", NewTemplate :=False
    With Selection.Font
        .Name = "Times New Roman"
        .Size = 16
        .Bold = False
        .Italic = False
        .Underline = wdUnderlineNone
        .StrikeThrough = False
        .DoubleStrikeThrough = False
        .Outline = False
        .Emboss = False
        .Shadow = False
        .Hidden = False
        .SmallCaps = False
        .AllCaps = False
        .ColorIndex = wdAuto
        .Engrave = False
        .Superscript = False
        .Subscript = False
        .Spacing = 0
        .Scaling = 100
        .Position = 0
        .Kerning = 0
        .Animation = wdAnimationNone
    End With
    Selection.TypeText Text:="Überschrift"
    Selection.HomeKey Unit:=wdLine
End Sub
```

Aus diesem Quellcode-Haufen filtern wir uns erst einmal die relevanten Daten heraus:

```
Sub Makro1()
  Documents.Add
  Selection.Font.Size = 16
  Selection.TypeText Text:="Überschrift"
End Sub
```

Das sieht doch schon viel freundlicher aus, das Resultat beider Makros ist dasselbe. Kopieren Sie nun den Quellcode in Ihre C#-Anwendung. Erster Schritt ist jetzt das Erzeugen einer Word-Instanz bzw. einer *Word.Application*-Instanz:

```
using Word = Microsoft.Office.Interop.Word;
...
private void button1_Click(object sender, EventArgs e)
{
```

BEISPIEL 34.35: Transformieren eines aufgezeichneten Word-Markros in C#

```
    var myWord = new Word.Application();
```

Word muss extra einblendet werden:

```
    myWord.Visible = true;
```

Uns hier kommen die eigentlichen Anweisungen:

```
    myWord.Documents.Add();
    myWord.Selection.Font.Size = 16;
    myWord.Selection.TypeText("Überschrift");
}
```

Der ehemalige Makrocode ist fett hervorgehoben.

HINWEIS: Das obige C#-Listing ist erst ab C# 4.0 lauffähig, in älteren Versionen müssen Sie sich damit abfinden, dass Sie immer alle Parameter an die jeweiligen Methoden übergeben müssen. Dies erfordert zum Teil einen beträchtlichen Mehraufwand und ein intensives Studium der VBA-Hilfe.

BEISPIEL 34.36: Umsetzung ab C# 2008

C#

```
using Microsoft.Office.Interop.Word;
...
      ApplicationClass myWord = new ApplicationClass();
      object n = System.Reflection.Missing.Value;
      myWord.Visible = true;
      myWord.Documents.Add(ref n, ref n, ref n, ref n);
      myWord.Selection.Font.Size = 16;
      myWord.Selection.TypeText("Überschrift");
```

HINWEIS: Beachten Sie, dass in diesem Fall alle Methoden-Parameter mittels Referenz (*ref*) zu übergeben sind.

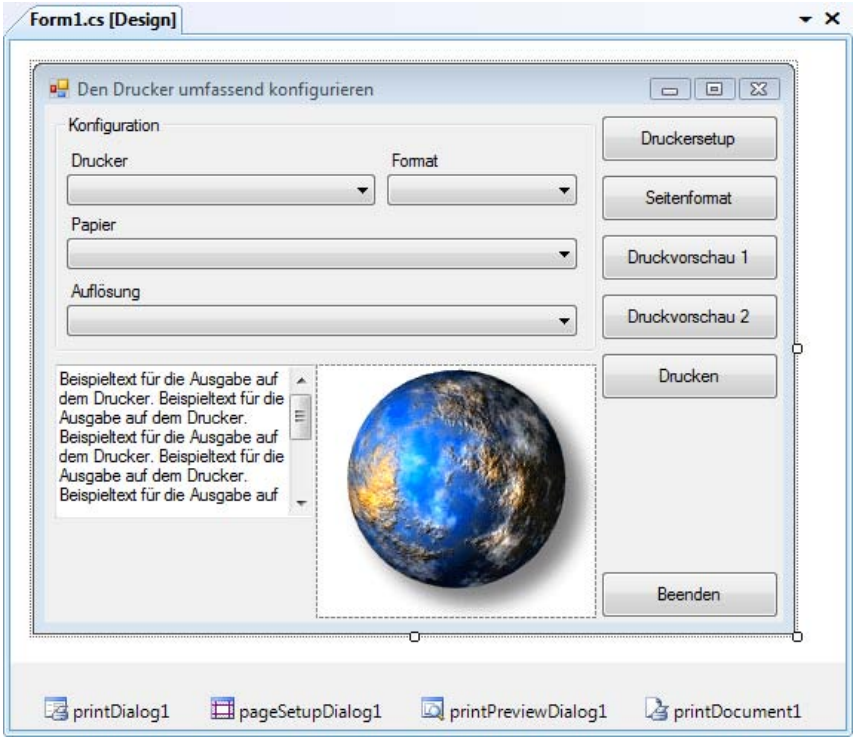
34.6 Praxisbeispiele

34.6.1 Den Drucker umfassend konfigurieren

Das Ziel dieses Beispiels ist eine umfassende Darstellung des Zusammenspiels der einzelnen Drucker-Komponenten sowie deren Konfiguration per Code bzw. per Dialogbox. Insgesamt zehn Beispielseiten verdeutlichen die verschiedenen Möglichkeiten der Gestaltung des Druckbildes.

Oberfläche (Hauptformular Form1)

Entwerfen Sie eine Oberfläche entsprechend folgender Abbildung:

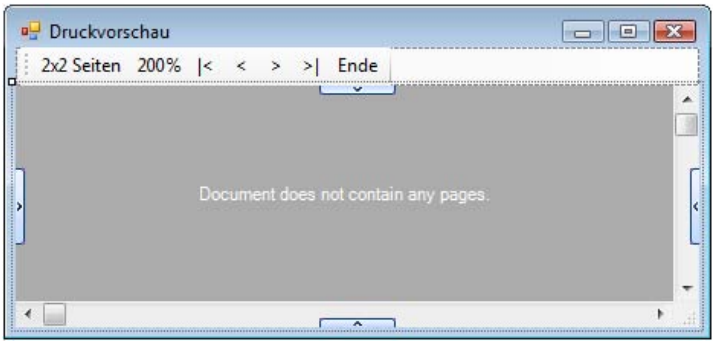


Verknüpfen Sie die vier nicht sichtbaren Komponenten (siehe unterer Bildrand) über die *Documents*-Eigenschaft mit dem *printDocument1*.

Sowohl die *TextBox* als auch die *PictureBox* dienen uns lediglich als Container für einen zu druckenden Text bzw. eine zu druckende Grafik.

Oberfläche (Druckvorschau Form2)

Mit der folgenden Oberfläche wollen wir keinen Schönheitspreis gewinnen, es geht lediglich um die Darstellung des Grundprinzips. Welche Komponenten Sie für die Oberflächengestaltung nutzen, bleibt Ihrer Fantasie überlassen. Wichtig ist vor allem das *PrintPreview*-Control:



Quelltext (Form1)

```
using System.Drawing.Printing;

public partial class Form1 : Form
{
```

Eine globale Variable erleichtert uns die Anzeige bzw. den Druck der richtigen Seite:

```
    private int page;
```

Die folgende Routine aktualisiert die *ComboBox*en nach Änderungen über die Standarddialoge:

```
    private void aktualisieren()
    {
```

Der aktuelle Drucker:

```
        comboBox1.Text = printDocument1.PrinterSettings.PrinterName;
```

Die verschiedenen Papierformate:

```
        comboBox2.Items.Clear();
        foreach (PaperSize ps in printDocument1.PrinterSettings.PaperSizes)
          comboBox2.Items.Add(ps);
        comboBox2.Text = printDocument1.DefaultPageSettings.PaperSize.ToString();
```

Die Seitenausrichtung:

```
        if (printDocument1.DefaultPageSettings.Landscape)
          comboBox3.SelectedIndex = 0;
         else
          comboBox3.SelectedIndex = 1;
```

Die Druckauflösung:

```
        comboBox4.Items.Clear();
        foreach (PrinterResolution res in printDocument1.PrinterSettings.PrinterResolutions)
                  comboBox4.Items.Add(res);
        comboBox4.Text = printDocument1.DefaultPageSettings.PrinterResolution.ToString();
    }
```

Beim Programmstart füllen wir zunächst *comboBox1* mit den Namen der verfügbaren Drucker und aktualisieren die Anzeige:

```
    private void Form1_Load(object sender, EventArgs e)
    {
       foreach(String s in PrinterSettings.InstalledPrinters)
            comboBox1.Items.Add(s);
       aktualisieren();
    }
```

Die Anzeige des Standard-Druckerdialogs:

```
    private void Button2_Click(object sender, EventArgs e)
    {
```

```
      printDialog1.ShowDialog();
      aktualisieren();
   }
```

Das Einrichten der Seite (Fehler bei der Umrechnung beachten!):

```
   private void Button3_Click(object sender, EventArgs e)
   {
       pageSetupDialog1.PageSettings.Margins.Left =
               (int) (pageSetupDialog1.PageSettings.Margins.Left * 2.54);
       pageSetupDialog1.PageSettings.Margins.Top =
               (int) (pageSetupDialog1.PageSettings.Margins.Top * 2.54);
       pageSetupDialog1.PageSettings.Margins.Right =
               (int) (pageSetupDialog1.PageSettings.Margins.Right * 2.54);
       pageSetupDialog1.PageSettings.Margins.Bottom =
               (int) (pageSetupDialog1.PageSettings.Margins.Bottom * 2.54);
       pageSetupDialog1.ShowDialog();
       aktualisieren();
   }
```

Start des Druckvorgangs bzw. der Druckvorschau:

```
   private void printDocument1_BeginPrint(object sender,
                                       System.Drawing.Printing.PrintEventArgs e)
   {
      page = 1;
      printDocument1.DocumentName = "Mein erstes Testdokument";
   }
```

Das eigentliche Drucken der Seiten passiert wie immer im *PrintPage*-Event unseres *PrintDocument*-Objekts:

```
   private void printDocument1_PrintPage(object sender,
                                      System.Drawing.Printing.PrintPageEventArgs e)
   {
```

Eine Zufallszahl für optische Spielereien:

```
      Random rnd = new Random();
```

Einen *Pen* definieren:

```
      Pen p = new Pen(System.Drawing.Color.Black, 1);
```

Eine Variable für den einfacheren Zugriff auf das *Graphics*-Objekt:

```
      Graphics g = e.Graphics;
```

Die aktuell zu druckende Seite:

```
      int printpage = 0;
```

Umschalten in Millimeter:

```
      g.PageUnit = GraphicsUnit.Millimeter;
```

Berücksichtigung des Druckbereichs:

```
switch (e.PageSettings.PrinterSettings.PrintRange)
{
    case PrintRange.SomePages:
         printpage = page + e.PageSettings.PrinterSettings.FromPage - 1;
         break;
    case PrintRange.AllPages:
         printpage = page;
         break;
}
```

Drucken der jeweiligen Seite (1 bis 10):

```
switch (printpage)
{
     case 1:
```

Ein paar Rechtecke (10 x 10 cm):

```
          g.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
          g.FillRectangle(new SolidBrush(Color.Green), 40, 40, 100, 100);
          g.FillRectangle(new SolidBrush(Color.Yellow), 50, 50, 100, 100);
          g.FillRectangle(new SolidBrush(Color.Cyan), 60, 60, 100, 100);
          g.FillRectangle(new SolidBrush(Color.Red), 70, 70, 100, 100);
         break;
     case 2:
```

Einige Linien auf Seite 2:

```
          g.DrawLine(new Pen(Color.Black, 10), 50, 100, 150, 200);
         g.DrawLine(new Pen(Color.Black, 10), 50, 200, 150, 100);
         break;
     case 3:
```

Ausgabe der Grafik in Originalgröße:

```
          g.DrawString("Grafik 100%", new Font("Arial", 10, FontStyle.Bold,
                              GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
          g.DrawImage(pictureBox1.Image, 50, 100);
          break;
     case 4:
```

Skalieren der Grafik auf 10 cm Breite:

```
          g.DrawString("Grafik 10cm breit", new Font("Arial", 10, FontStyle.Bold,
                   GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
          g.DrawImage(pictureBox1.Image, 50, 100, 100,
                   pictureBox1.Image.Height * 100 % pictureBox1.Image.Width);
          g.DrawRectangle(new Pen(Color.Black, 0.1f), 50, 100, 100,
                     pictureBox1.Image.Height * 100 % pictureBox1.Image.Width);
         break;
     case 5:
```

Anzeige der Seitenränder:

```
            g.DrawString("Seitenränder", new Font("Arial", 10, FontStyle.Bold,
                                GraphicsUnit.Millimeter), Brushes.Black, 70, 50);
            g.PageUnit = GraphicsUnit.Display;
            g.DrawRectangle(new Pen(Color.Black), e.MarginBounds);
            g.PageUnit = GraphicsUnit.Millimeter;
            break;
        case 6:
```

Ausgabe von Text (linksbündig):

```
            RectangleF rect = new RectangleF();
            rect = RectangleF.op_Implicit(e.MarginBounds);
            g.PageUnit = GraphicsUnit.Display;
            g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                                GraphicsUnit.Millimeter), Brushes.Black, rect);
            g.PageUnit = GraphicsUnit.Millimeter;
            break;
        case 7:
```

Ausgabe von Text (zentriert):

```
            RectangleF rect1 = new RectangleF();
            StringFormat format = new StringFormat();
            format.Alignment = StringAlignment.Center;
            rect1 = RectangleF.op_Implicit(e.MarginBounds);
            g.PageUnit = GraphicsUnit.Display;
            g.DrawString(TextBox1.Text, new Font("Arial", 10, FontStyle.Bold,
                            GraphicsUnit.Millimeter), Brushes.Black, rect1, format);
            g.PageUnit = GraphicsUnit.Millimeter;
            break;
        case 8:
```

Ausgabe von zufälligen Linien über den gesamten Blattbereich:

```
            g.DrawString("Zufallslinien ohne Clipping", new Font("Arial", 10,
                        FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White, 70, 50);
            g.PageUnit = GraphicsUnit.Display;
            for (int i = 0; i <= 500; i++)
                        g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                            rnd.Next(e.PageBounds.Height));
            break;
        case 9:
```

HINWEIS: Vergleichen Sie den Ausdruck mit der Druckvorschau werden Sie feststellen, dass die Druckvorschau die physikalischen Seitenränder nicht berücksichtigt.

Berücksichtigung der Seitenränder bei der Druckausgabe:

```
            g.DrawString("Zufallslinien mit Clipping", new Font("Arial", 10,
```

```
                        FontStyle.Bold, GraphicsUnit.Millimeter), Brushes.White, 70, 50);
            g.PageUnit = GraphicsUnit.Display;
            g.SetClip(e.MarginBounds);
            for (int i = 0; i <=500; i++)
                g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                    rnd.Next(e.PageBounds.Height));
            break;
        case 10:
```

Berücksichtigung der Seitenränder sowie Verschieben des Offsets bei der Druckausgabe:

```
            g.PageUnit = GraphicsUnit.Display;
            g.SetClip(e.MarginBounds);
            g.TranslateTransform(e.MarginBounds.Left, e.MarginBounds.Top);
            for (int i = 0; i <= 500; i++)
                      g.DrawLine(p, 0, 0, rnd.Next(e.PageBounds.Width),
                                          rnd.Next(e.PageBounds.Height));
            break;
    }
```

Seitennummer einblenden:

```
    g.DrawString("Seite : " + printpage, new Font("Arial", 10, FontStyle.Bold,
                                  GraphicsUnit.Millimeter), Brushes.Red, 10, 10);
```

Vorbereiten der nächsten Seite:

```
    page++;
```

Berücksichtigung des Druckbereichs:

```
    switch (e.PageSettings.PrinterSettings.PrintRange)
    {
        case PrintRange.SomePages:
            e.HasMorePages = (printpage < e.PageSettings.PrinterSettings.ToPage);
            break;
        case PrintRange.AllPages:
            e.HasMorePages = (page < 12);
            break;
    }
}
```

Aktuellen Drucker wechseln:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
   printDocument1.PrinterSettings.PrinterName = comboBox1.Text;
   aktualisieren();
}
```

Seitenausrichtung ändern:

```
private void comboBox3_SelectedIndexChanged(object sender, EventArgs e)
{
```

```
        printDocument1.DefaultPageSettings.Landscape = (comboBox3.SelectedIndex == 0);
        aktualisieren();
    }
```

Papierformat ändern:

```
    private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PaperSize =
                            printDocument1.PrinterSettings.PaperSizes[comboBox2.SelectedIndex];
    }
```

Druckauflösung ändern:

```
    private void comboBox4_SelectedIndexChanged(object sender, EventArgs e)
    {
        printDocument1.DefaultPageSettings.PrinterResolution =
              printDocument1.PrinterSettings.PrinterResolutions[comboBox4.SelectedIndex];
    }
```

Druckvorschau anzeigen (Vollbild):

```
    private void Button1_Click(object sender, EventArgs e)
    {
         printPreviewDialog1.WindowState = FormWindowState.Maximized;
         printPreviewDialog1.ShowDialog();
    }
```

Die eigene Druckvorschau anzeigen:

```
    private void Button4_Click(object sender, EventArgs e)
    {
         Form2 f2 = new Form2();
         f2.printPreviewControl1.Document = printDocument1;
         f2.ShowDialog();
    }
```

Den Druckvorgang starten:

```
    private void Button5_Click(object sender, EventArgs e)
    {
         printDocument1.Print();
    }
}
```

Quelltext (Form2)

Im Formular *Form2* geht es im Wesentlichen nur um die Konfiguration der *PrintPreviewControl*-Komponente.

Die Navigation zwischen den Seiten:

```
public partial class Form2 : Form
{
```

```
    ...
```

Nächste Seite:

```
        private void toolStripButton4_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage++;
        }
```

Vorhergehende Seite:

```
        private void toolStripButton3_Click(object sender, EventArgs e)
        {
            if (printPreviewControl1.StartPage > 0) printPreviewControl1.StartPage--;
        }
```

Erste Seite:

```
        private void toolStripButton6_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 0;
        }
```

Letzte Seite (setzen Sie einfach einen Wert, der groß genug ist):

```
        private void toolStripButton7_Click(object sender, EventArgs e)
        {
            printPreviewControl1.StartPage = 999;
        }
```

Seite auf 200 Prozent skalieren:

```
        private void toolStripButton2_Click(object sender, EventArgs e)
        {
            printPreviewControl1.AutoZoom = false;
            printPreviewControl1.Zoom = 200;
        }
```

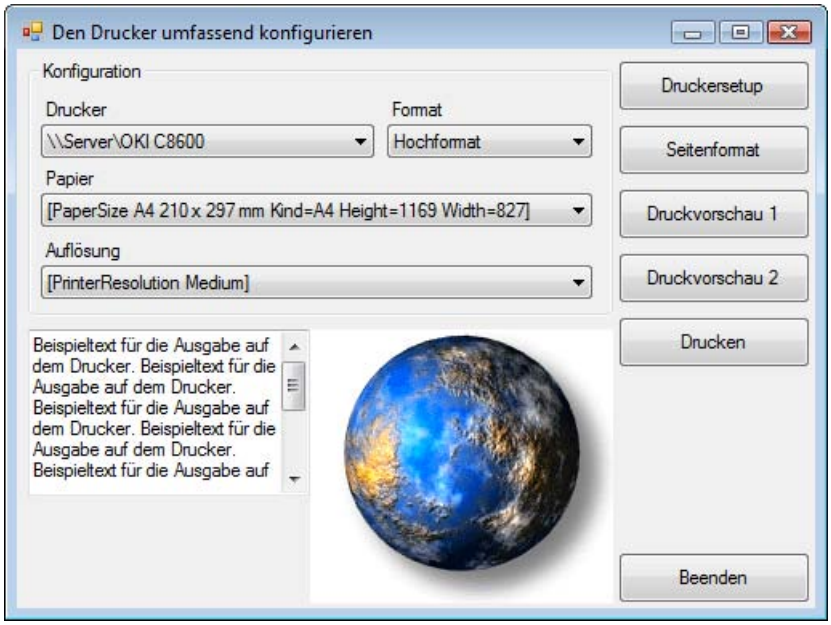
Vier Seiten gleichzeitig anzeigen (eingepasst in die Komponente):

```
        private void toolStripButton1_Click(object sender, EventArgs e)
        {
            printPreviewControl1.Columns = 2;
            printPreviewControl1.Rows = 2;
            printPreviewControl1.AutoZoom = true;
        }
}
```

Test

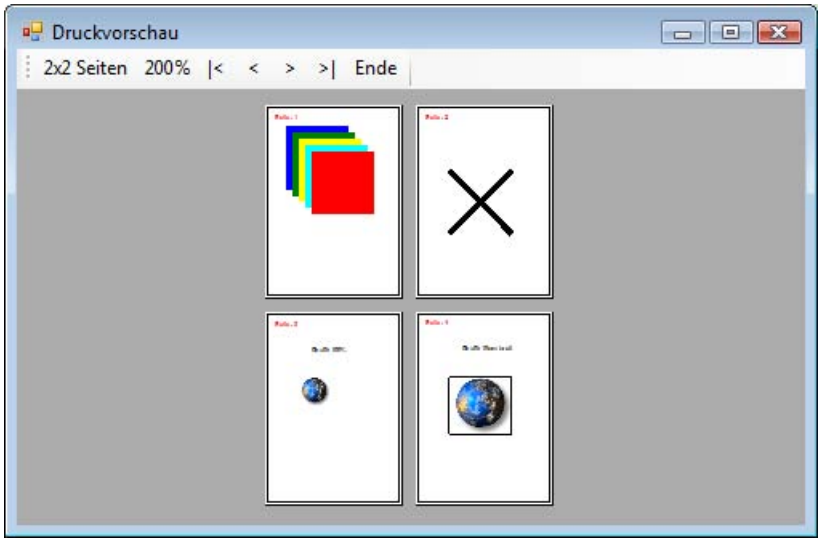
Nach dem Programmstart sollten alle Druckerparameter korrekt in den *ComboBoxen* angezeigt werden.

HINWEIS: Testen Sie, was passiert, wenn Sie Änderungen in den *ComboBoxen* bzw. mit Hilfe der Druckerdialoge vornehmen.



Nun haben Sie die Möglichkeit, sich die 10 verschiedenen Druckseiten in einer der beiden Druckvorschauen zu betrachten oder zu Papier zu bringen:

Als Beispiel hier unsere "selbst gebastelte" Druckvorschau in Aktion:



34.6.2 Diagramme mit dem Chart-Control drucken

Mit diesem Beispiel wollen wir uns nicht um das recht einfache direkte Ausdrucken eines Diagramms per *Chart.Printing.Print*-Methode kümmern, sondern das Diagramm im Rahmen eines eigenen Berichts ausgeben. Auf diese Weise haben Sie die Möglichkeit, das Diagramm mit weiteren Deckblättern, Anmerkungen oder auch Tabellen auszugeben.

Oberfläche

Fügen Sie in ein Windows Forms-Form zunächst eine *PrintDocument* und eine *PrintPreviewDialog*-Komponente ein. Verknüpfen Sie beide Komponenten über die *Document*-Eigenschaft von *PrintPreviewDialog1*.

Zusätzlich brauchen Sie noch ein *Chart*-Control und zwei Schaltflächen (siehe Laufzeitansicht). Die Konfiguration des *Chart*-Controls nehmen Sie per Code vor, damit ist der Oberflächenentwurf abgeschlossen.

Quelltext

Erweitern Sie den Quellcode von *Form1* um folgenden Anweisungen:

```
    public partial class Form1 : Form
    {
```

Unser Seitenzähler für die Druckausgabe:

```
        private int page;
...
```

Mit dem Klick auf die erste Schaltfläche generieren wir zunächst zwei Diagramme, die wir im Weiteren ausgeben wollen:

```
        private void button1_Click(object sender, EventArgs e)
        {
            chart1.Series[0].Name = "Umsätze TFT";
            chart1.Series[0].Points.AddXY(2007, 10);
            chart1.Series[0].Points.AddXY(2008, 25);
            chart1.Series[0].Points.AddXY(2009, 75);
            chart1.Series[0].Points.AddXY(2010, 150);
            chart1.ChartAreas.Add("Area2");
            chart1.Series.Add("Umsätze Telefone");
            chart1.Series[1].ChartArea = "Area2";
            chart1.Series[1].Points.AddXY(2007, 150);
            chart1.Series[1].Points.AddXY(2008, 65);
            chart1.Series[1].Points.AddXY(2009, 15);
            chart1.Series[1].Points.AddXY(2010, 5);
        }
```

Mit Beginn des Ausdruckens setzen wir unseren Seitenzähler zurück:

```
        private void printDocument1_BeginPrint(object sender,
                                               System.Drawing.Printing.PrintEventArgs e)
```

```
        {
            page = 1;
            printDocument1.DocumentName = "Mein erstes Testdokument";
        }
```

Die eigentliche Druckroutine:

```
        private void printDocument1_PrintPage(object sender,
                                              System.Drawing.Printing.PrintPageEventArgs e)
        {
```

Umschalten in die für einen Drucker praktikableren Millimeter:

```
            Graphics g = e.Graphics;
            g.PageUnit = GraphicsUnit.Millimeter;
```

Hier unterscheiden wir die einzelnen Seiten:

```
            switch (page)
            {
```

Unser "Deckblatt":

```
                case 1:
                    g.FillRectangle(new SolidBrush(Color.Blue), 30, 30, 100, 100);
                    break;
```

Das eigentliche Diagramm aus dem *Chart*-Control:

```
                case 2:
                    chart1.Printing.PrintPaint(g, new Rectangle(10, 10, 180, 250));
                    break;
            }
```

Nächste Seite auswählen:

```
            page++;
```

Es sind maximal zwei Seiten vorhanden:

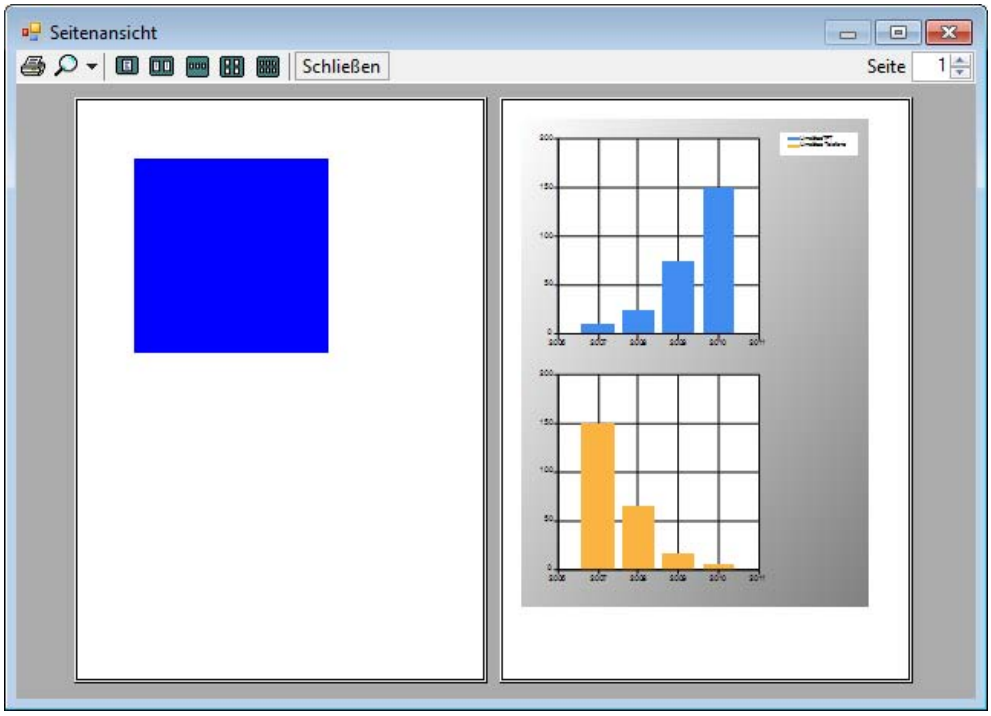
```
            e.HasMorePages = page < 3;
        }
```

Anzeige der Druckvorschau:

```
        private void button2_Click(object sender, EventArgs e)
        {
            printPreviewDialog1.ShowDialog();
        }
    }
}
```

Test

Nach dem Start erzeugen Sie zunächst über die obere Schaltfläche die Diagramme, nachfolgend können Sie die Druckvorschau aufrufen und sich vom Ergebnis überzeugen:

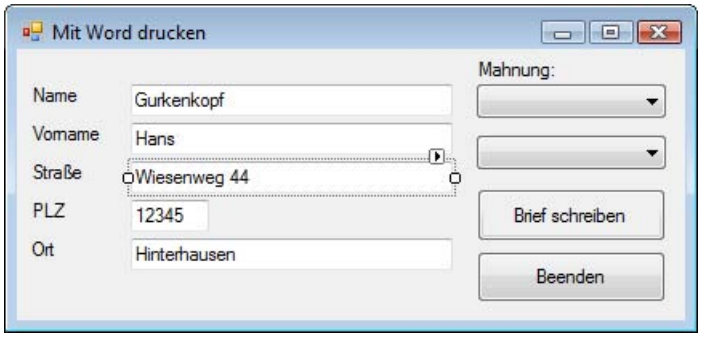


34.6.3 Druckausgabe mit Word

Eines der "dankbarsten Opfer" für OLE-Automation ist nach wie vor Word für Windows. Unser Beispiel zeigt Ihnen, wie Sie aus einem C#-Programm heraus ein neues Word-Dokument erstellen, Kopf- und Fußzeilen einfügen und Daten übertragen (das Beispiel lässt sich problemlos so anpassen, dass die Daten statt aus den Eingabefeldern gleich aus einer Datenbank kommen).

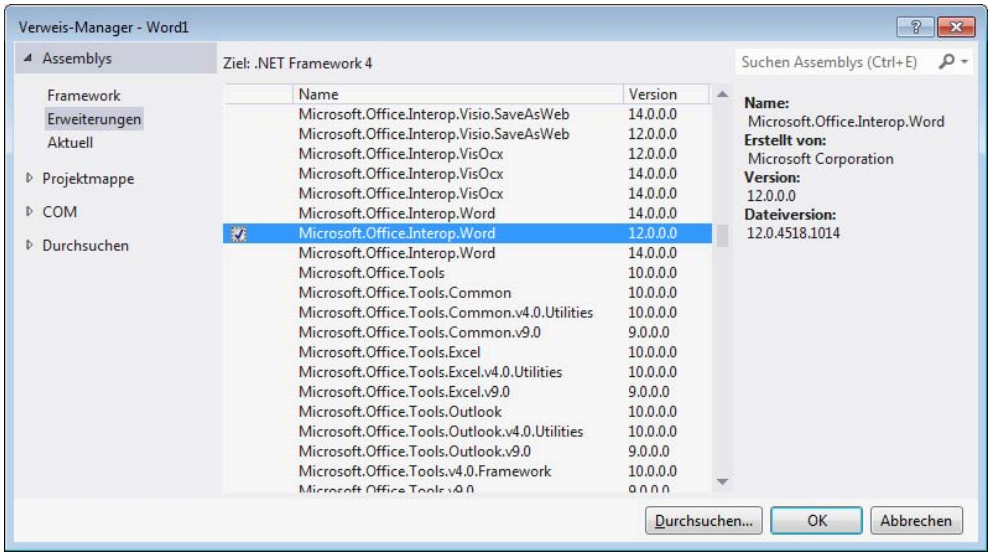
Oberfläche

Den Grundaufbau können Sie der folgenden Abbildung entnehmen:



In der oberen *ComboBox* finden sich drei Einträge: "1. Mahnung" ... "3. Mahnung", die Sie im Eigenschaftenfenster über die *Items*-Auflistung hinzufügen, in der unteren *ComboBox* wird zwischen "Herr" und "Frau" unterschieden.

Damit Sie problemlos die Word-Objekte und -Konstanten verwenden können, müssen Sie noch einen Verweis auf die *Microsoft.Office.Interop.Word*-Assembly (eine *PIA = Primary Interop Assembly*) einrichten (*Projekt|Verweis hinzufügen...*).



Quelltext

```
using Word = Microsoft.Office.Interop.Word;

public partial class Form1 : Form
{
```

Es geht los:

```
        private void button1_Click(object sender, EventArgs e)
        {
```

Grundlage für die Verbindung zu Word ist eine implizit typisierte lokale Variable vom Typ *ApplicationClass*:

```
            var wordapp = new Word.Application();
```

Der Ablauf ist mit wenigen Worten erklärt: Nach der Initialisierung der Variablen können Sie alle Methoden des *Application*-Objekts verwenden. Bevor Sie lange in der Word-Dokumentation herumstochern, ist es sinnvoller, ein Word-Makro aufzuzeichnen und dieses entsprechend zu modifizieren. Zum einen haben Sie gleich die korrekte Syntax, zum anderen sparen Sie sich jede Menge Arbeit.

Bei Problemen kneifen wir an dieser Stelle:

```
if (wordapp == null)
{
    MessageBox.Show("Konnte keine Verbindung zu Word herstellen!");
    return;
}
```

Word sichtbar machen (standardmäßig wird Word nicht angezeigt):

```
wordapp.Visible = true;
```

Ein neues Dokument erzeugen:

```
wordapp.Documents.Add();
if (wordapp.ActiveWindow.View.SplitSpecial != 0)
    wordapp.ActiveWindow.Panes[2].Close();
if (((int) wordapp.ActiveWindow.ActivePane.View.Type == 1)|
              ((int) wordapp.ActiveWindow.ActivePane.View.Type == 2)|
                        ((int)wordapp.ActiveWindow.ActivePane.View.Type == 5))
                   wordapp.ActiveWindow.ActivePane.View.Type =
                            Word.WdViewType.wdPrintView;
```

Kopfzeile erzeugen:

```
wordapp.ActiveWindow.ActivePane.View.SeekView =
        Word.WdSeekView.wdSeekCurrentPageHeader;
wordapp.Selection.Font.Name = "Times New Roman";
wordapp.Selection.Font.Size = 12;
wordapp.Selection.Font.Bold = 1;
wordapp.Selection.ParagraphFormat.Alignment =
        Word.WdParagraphAlignment.wdAlignParagraphCenter;
wordapp.Selection.TypeText(
        "Kohlenhandel Brikett-GmbH & Co.-KG. - Holzweg 16 - 54633 Steinhausen");
```

Fußzeile erzeugen:

```
wordapp.ActiveWindow.ActivePane.View.SeekView =
       Word.WdSeekView.wdSeekCurrentPageFooter;
wordapp.Selection.TypeText(
      "Bankverbindung: Stadtsparkasse Steinhausen BLZ 123456789   KtoNr. " +
      "782972393243");
```

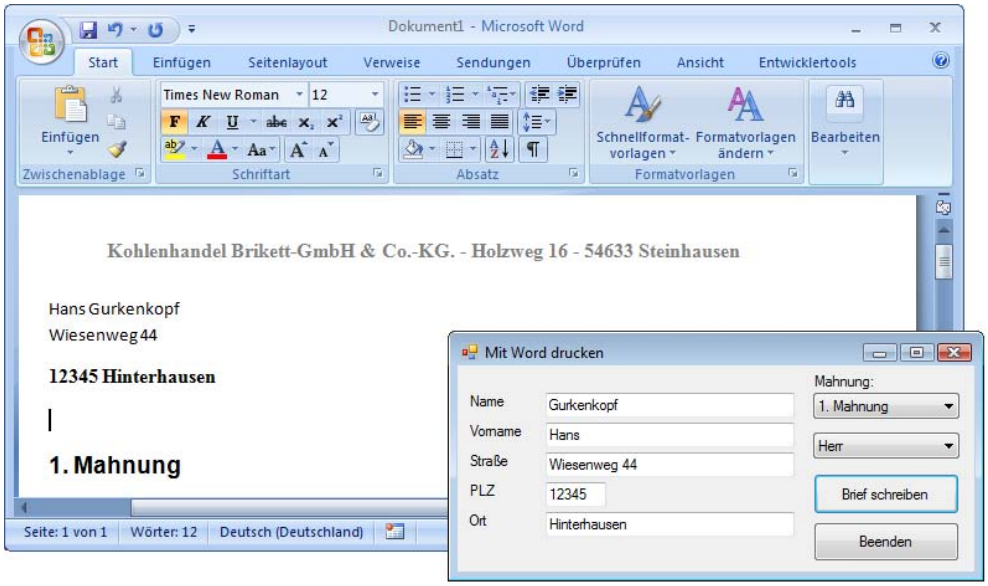
In den Textteil wechseln und die Adresse eintragen:

```
wordapp.ActiveWindow.ActivePane.View.SeekView = Word.WdSeekView.wdSeekMainDocument;
wordapp.Selection.TypeText(textBox2.Text + " " + textBox1.Text);
wordapp.Selection.TypeParagraph();
wordapp.Selection.TypeText(textBox3.Text);
wordapp.Selection.TypeParagraph();
wordapp.Selection.Font.Name = "Times New Roman";
wordapp.Selection.Font.Size = 12;
wordapp.Selection.Font.Bold = 1;
wordapp.Selection.TypeText(textBox4.Text + " " + textBox5.Text);
```

```
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Arial";
            wordapp.Selection.Font.Size = 14;
            wordapp.Selection.Font.Bold = 1;
            wordapp.Selection.TypeText(comboBox1.Text);
            wordapp.Selection.TypeParagraph();
            wordapp.Selection.Font.Name = "Times New Roman";
            wordapp.Selection.Font.Size = 12;
            wordapp.Selection.Font.Bold = 1;

            if (comboBox2.SelectedIndex == 0)
                wordapp.Selection.TypeText("Sehr geehrter Herr " + textBox1.Text);
            else
                wordapp.Selection.TypeText("Sehr geehrte Frau " + textBox1.Text);
        }
    }
```

Test

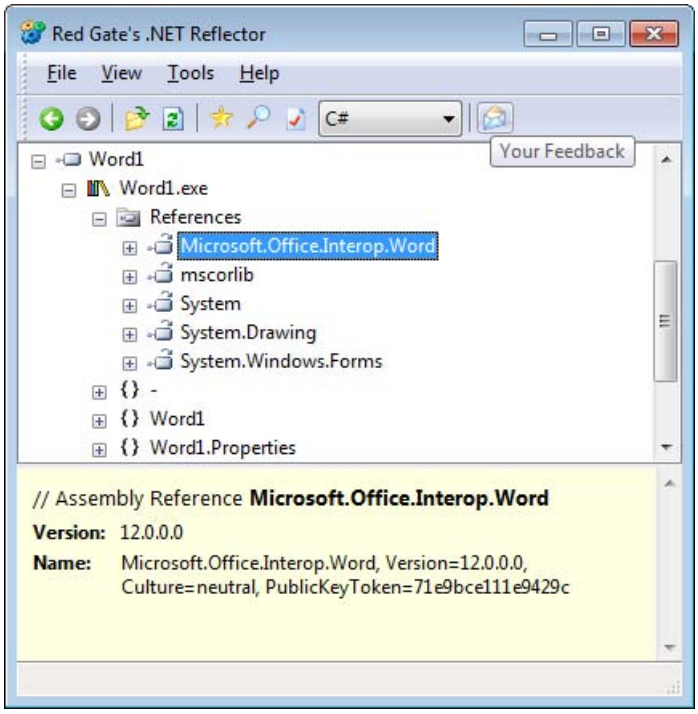
Starten Sie das Programm, füllen Sie die Maske aus und übertragen Sie die Daten in ein Word-Dokument!



Anmerkung ab .NET 4.0

Vielleicht hat mancher Leser bei Verwendung der mittlerweile berühmt berüchtigten PIA[1]s schlechte Erfahrungen gesammelt. So erscheint schnell mal eine Fehlermeldung wenn auf dem Ziel-PC nicht die entsprechenden Assemblies installiert sind.

Hintergrund derartiger "Problemchen" ist die Tatsache, dass beim Kompilieren eines Projekts mit eingebundenen PIAs nur Verweise auf eben diese Assemblies eingebunden werden, wie es auch folgende Abbildung (erstellt mittels .NET-Reflector) zeigt:



Die Assemblies müssen also gegebenenfalls mitgegeben werden, was den Umfang Ihres Projekts jedoch schnell aufblähen kann.

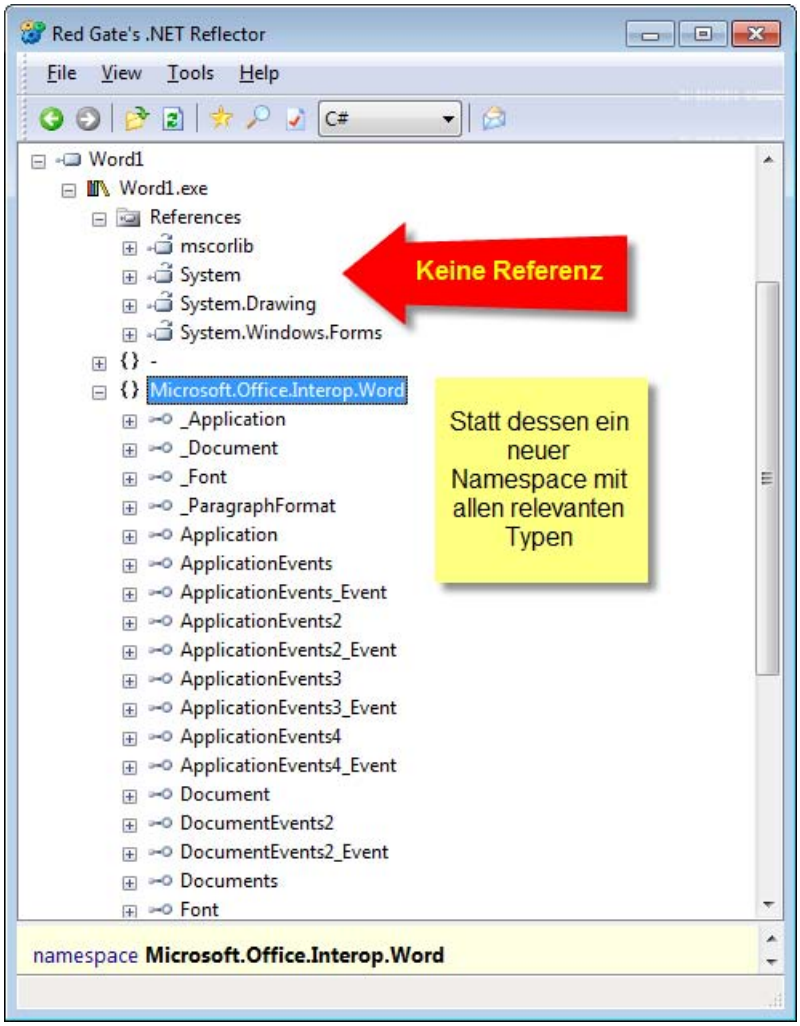
Seit .NET 4 besteht die Möglichkeit, die verwendeten Interop-Typen in Ihre Assembly einzubetten. Damit entfällt auch die Weitergabe bzw. Installation der PIAs auf dem Ziel-PCs.

HINWEIS: Stellen Sie alte Projekte um, müssen Sie auch die Ziel-Plattform auf .NET 4 anpassen, andernfalls steht Ihnen dieses Feature nicht zur Verfügung!

Zum Einbetten genügt es, wenn Sie die Eigenschaft "Interop-Typen einbetten" der jeweiligen PIA auf *true* setzen (dies ist bei neuen Projekten standardmäßig der Fall).

[1] *Primary Interop Assembly*

Ein erneuter Blick mit dem .NET-Reflector in die Anwendung zeigt uns jetzt, dass keine Verweise mehr eingebunden werden, statt dessen finden Sie einen neuen Namenspace und die entsprechenden Typen in Ihrer Anwendung vor:



Wer jetzt befürchtet, dass sich die Anwendung massiv aufbläht, liegt falsch, es werden wirklich nur die unmittelbar benötigten Typen in die Assembly aufgenommen.

Kapitel 35

Windows Forms-Datenbindung

Sie haben in den Kapiteln 24 und 25 bereits die Grundlagen von ADO.NET kennen gelernt und wissen, wie man Datenbanken abfragen und aktualisieren kann. Um Ein- und Ausgaben zu realisieren, hatten wir dort bereits mit einfacher Datenbindung gearbeitet (meist unter Verwendung des *DataGridView*). An dieser Stelle wollen wir uns dieser Thematik im Detail widmen.

HINWEIS: Wesentlich umfassender und detaillierter gehen wir auf diese Thematik in unserem Buch [Visual C# 2012 Datenbankprogrammierung] ein, dies betrifft vor allem die Arbeit mit *DatagridView* und *Chart*-Control.

35.1 Prinzipielle Möglichkeiten

Datenbindung ist ganz allgemein die Verknüpfung zwischen einer Steuerelementeigenschaft und einer Datenquelle. In Abhängigkeit von der Beantwortung der beiden Fragen

- "*Will ich die Datenbindung manuell oder mit Drag & Drop-Assistentenunterstützung programmieren?*" oder
- *"Sollen komplette Listen bzw. Tabelleninhalte oder nur einzelne Felder angebunden werden?*"

... kann man das Gebiet der Datenbindung grob in vier Bereiche aufteilen:

- Manuelle Datenbindung an einfache Datenfelder
- Manuelle Datenbindung an Listen/Tabelleninhalte
- Entwurfszeit-Datenbindung an ein typisiertes DataSet
- Drag&Drop-Datenbindung

Wer wenig Code schreiben will, wird weitestgehend die Hilfe der Assistenten in Anspruch nehmen. Der solide Handwerker, der lieber etwas mehr Code schreibt und dafür aber die volle Kontrolle über sein Programm behält, wird die manuelle Datenbindung bevorzugen.

35.2 Manuelle Bindung an einfache Datenfelder

Bestimmte Eigenschaften vieler Windows Forms-Controls lassen sich an eine Datenquelle binden. Damit ändert der Wert in der Datenquelle den Wert der gebundenen Eigenschaft und umgekehrt.

BEISPIEL 35.1: Ein DataSet *ds* enthält die Tabelle "Personal". Eine *TextBox* soll an das Feld "Nachname" angebunden werden.

C#

Fügen Sie von der Toolbox eine *BindingSource*-Komponente zum Formular hinzu.

```
bindingSource1.DataSource = ds;
bindingSource1.DataMember = "Personal";
```

Die *Text*-Eigenschaft der *TextBox* wird angebunden:

```
textBox1.DataBindings.Add("Text", bindingSource1, "Nachname");
```

Um die Datensätze weiterblättern zu können brauchen Sie nur noch eine *BindingNavigator*-Komponente hinzuzufügen, deren *BindingSource*-Eigenschaft Sie auf *bindingSource1* setzen.

35.2.1 BindingSource erzeugen

Die mit .NET 2.0 eingeführte *BindingSource* löste die veralteten (aber natürlich nach wie vor unterstützten) Klassen *BindingManagerBase* bzw. *CurrencyManager* ab. Eine *BindingSource* kapselt die Datenquelle des Formulars, sie schiebt sich quasi als zusätzliche Schicht zwischen Datenquelle und Anzeigecontrols. Mittels *DataSource*- bzw. *DataMember*-Eigenschaft wird eine *BindingSource* mit der Datenquelle verbunden.

BEISPIEL 35.2: Verschiedene Varianten zum Erzeugen einer *BindingSource* und ihrer Verbindung mit der Tabelle "Personal" eines *DataSet*-Objekts *ds*.

C#

```
BindingSource bs = new BindingSource();
bs.DataSource = ds;
bs.DataMember = "Personal";
```

oder

```
BindingSource bs = new BindingSource(ds, "Personal");
```

oder

```
DataTable dt = ds.Tables["Personal"];
BindingSource bs = new BindingSource();
bs.DataSource = dt;
```

oder

```
DataView dv = ds.Tables["Personal"].DefaultView;
BindingSource bs = new BindingSource();
bs.DataSource = dv;
```

35.2.2 Binding-Objekt

Ein *Binding*-Objekt ermöglicht die einfache Bindung zwischen dem Wert einer Objekteigenschaft und dem Wert einer Steuerelementeigenschaft. Bei der Instanziierung sind drei Parameter zu übergeben:

- die zu bindende Eigenschaft des Controls (z.B. *Text*),
- die Datenquelle, an die zu binden ist (*BindingSource, DataSet*, *DataTable*, *DataView*),
- das Feld innerhalb der Datenquelle, das angebunden werden soll (z.B. *Vorname*).

BEISPIEL 35.3: Die Steuerelementeigenschaft *Text* wird an die Eigenschaft Geburtsdatum der Personal-Tabelle gebunden.

C#
```
BindingSource bs = new BindingSource(ds, "Personal");
Binding b1 = new Binding("Text", bs, "Geburtsdatum");
```

35.2.3 DataBindings-Collection

Die Datenanbindung für einfache Steuerelemente, wie z.B. *Label* oder *TextBox*, wird durch Hinzufügen von *Binding*-Objekten zur *DataBindings*-Auflistung des Steuerelements komplettiert. Der *Add*-Methode sind entweder ein komplettes *Binding*-Objekt oder aber dessen drei Argumente zu übergeben.

BEISPIEL 35.4: (Fortsetzung)

C#

Das im Beispiel 35.4 erzeugte *Binding*-Objekt wird zur *DataBindings*-Collection einer *TextBox* hinzugefügt:

```
textBox1.DataBindings.Add(b1);
```

Eine Überladung der *Add*-Methode, die ohne explizit erzeugtes *Binding*-Objekt auskommt:

```
textBox1.DataBindings.Add("Text", bs, "Geburtsdatum");
```

Bemerkungen

- Mit der *Control*-Eigenschaft können Sie das Steuerelement abrufen, zu dem die *DataBindings*-Collection gehört.
- Nachdem die Steuerelemente angebunden sind, werden lediglich die Werte der ersten Zeile der *DataTable* angezeigt, Möglichkeiten zum Navigieren bzw. Blättern sind noch nicht vorhanden.

35.3 Manuelle Bindung an Listen und Tabellen

Bei dieser komplexeren Form der Datenbindung wollen wir Steuerelemente, die mehrere Werte anzeigen können, an eine Liste von Werten binden. Die dafür am häufigsten verwendeten Steuerelemente sind *DataGridView*, *ComboBox* oder *ListBox*.

35.3.1 DataGridView

Das *DataGridView* ist ein sehr leistungsfähiges Datengitter-Steuerelement, welches wir in den Kapiteln 10 und 11 bereits sehr häufig für die Anzeige von Tabelleninhalten benutzt haben.

BEISPIEL 35.5: Anzeige der Personal-Tabelle im *DataGridView*

C#

```
dataGridView1.DataSource = ds;
dataGridView1.DataMember = "Personal";
```

oder

```
BindingSource bs = new BindingSource(ds, "Personal");
dataGridView1.DataSource = bs;
```

35.3.2 Datenbindung von ComboBox und ListBox

Häufig werden *ComboBox* und *ListBox* zum Implementieren sogenannter "Nachschlagefunktionalität" bei *DataTable*s (oder *DataView*s) eingesetzt, zwischen denen eine Master-Detail-Relation besteht. Um die *ComboBox*/*ListBox* mit der Master-Tabelle zu verknüpfen, muss zunächst die *SelectedValue*Eigenschaft an den in der Mastertabelle enthaltenen Fremdschlüssel angebunden werden. Anschließend werden den *DataSource*-, *DisplayMember*- und *ValueMember*-Eigenschaften die entsprechenden Spalten der Detailtabelle zugewiesen.

BEISPIEL 35.6: Datenbindung von *ComboBox*

C#

Die Tabellen *Bestellungen* und *Personal* der *Nordwind*-Datenbank sind durch eine Master-Detail-Beziehung verknüpft. In der *ComboBox* soll der zur aktuellen Bestellung gehörige *Nachname* aus der *Personal*-Tabelle angezeigt werden.

Verbinden der *ComboBox* mit der Mastertabelle:

```
bindingSourceBest.DataSource = ds.Tables["Bestellungen"];
comboBox1.DataBindings.Add("SelectedValue", bindingSourceBest, "PersonalNr");
```

Anbinden der Detaildaten an die *ComboBox*:

```
bindingSourcePers.DataSource = ds.Tables["Personal"];

comboBox1.DataSource = bindingSourcePers;
comboBox1.DisplayMember = "Nachname";
comboBox1.ValueMember = "PersonalNr";
```

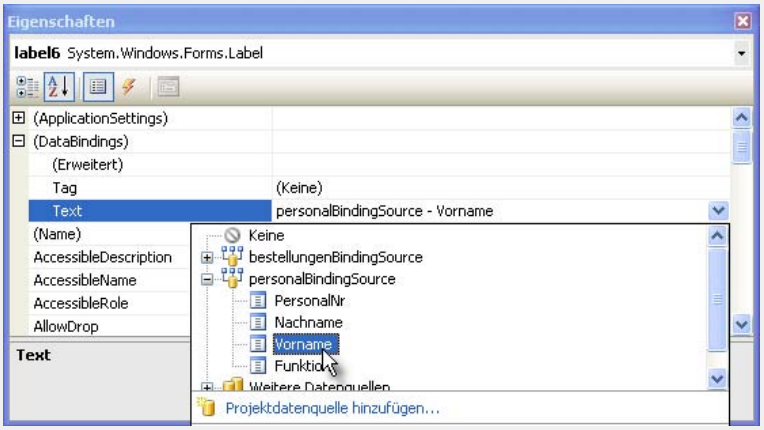
35.4 Entwurfszeit-Bindung an typisierte DataSets

Zwar basiert ADO.NET auf dem Prinzip der strikten Trennung der Benutzerschnittstelle von der Datenbank, doch gibt es trotzdem die Möglichkeit der Entwurfszeitanbindung der Steuerelemente. Allerdings muss dazu eine Datenquelle (typisiertes DataSet) vorhanden sein, welche nur mit Assistentenhilfe sinnvoll zu erstellen ist.

BEISPIEL 35.7: Entwurfszeitdatenbindung

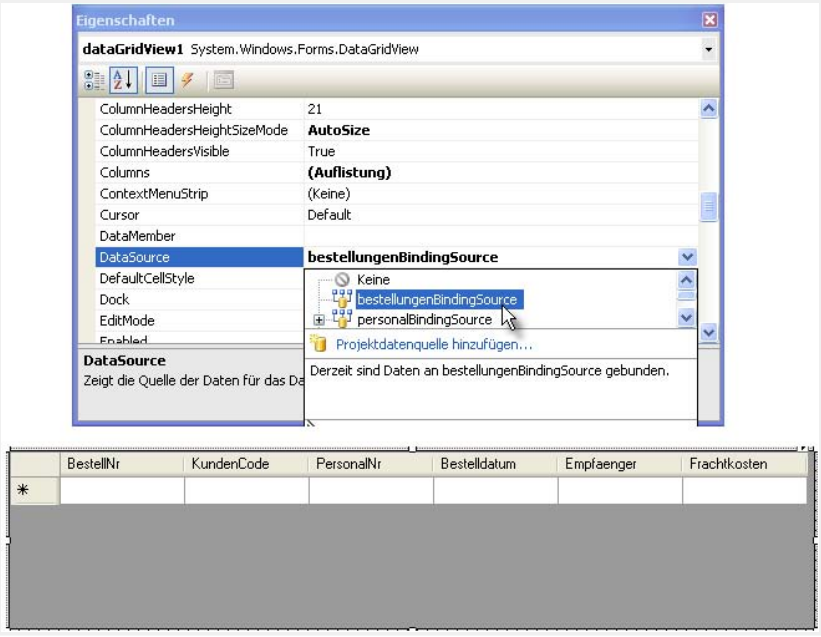
C#

Die folgende Abbildung zeigt, wie Sie über den *(DataBindings)*-Knoten im Eigenschaftenfenster die Datenbindung für ein *Label*-Steuerelement vornehmen. Die Datenfelder stehen dabei als *BindingSource*-Elemente zur Verfügung.



Auf analoge Weise realisieren Sie z.B. auch Entwurfszeit-Datenbindungen für *TextBox*, *ComboBox* und *ListBox* sowie mit dem *DataGridView*.

Ein *DataGridView* wird über eine *BindingSource* mit der Tabelle "Bestellungen" eines typisierten DataSets verbunden. Bereits zur Entwurfszeit zeigt das *DataGridView* die Datenstruktur.



35.5 Drag & Drop-Datenbindung

Unter der Voraussetzung, dass eine Datenquelle vorhanden ist, brauchen Sie nur noch per Drag & Drop komplette Tabellen aus dem Datenquellen-Fenster auf das Formular zu ziehen. Neben einer fertigen Eingabemaske (wahlweise Einzelkomponenten mit *BindingNavigator* oder als *DataGridView*) werden auch eine Unmenge von Datenzugriffskomponenten (*DataSet*, *BindingSource*, *TableAdapter*, ...) generiert und im Komponentenfach abgelegt.

HINWEIS: Eine komplette Anleitung finden Sie im PB 35.8.1 "Einrichten und Verwenden einer Datenquelle".

35.6 Navigations- und Bearbeitungsfunktionen

Für das Durchblättern der Datensätze sowie für Editieren, Hinzufügen und Löschen haben Sie hauptsächlich zwei Möglichkeiten:

- Sie können die verschiedenen Methoden der *BindingSource* verwenden oder
- Sie verwenden einen *BindingNavigator,* der die Methodenaufrufe kapselt.

35.6.1 Navigieren zwischen den Datensätzen

So wie das gute alte Recordset-Objekt aus den Zeiten vor .NET hat auch die *BindingSource* die Methoden *MoveNext*, *MovePrevious*, *MoveFirst* und *MoveLast.*

BEISPIEL 35.8: Bewegen zum ersten Datensatz

C#
```
BindingSource bs = new BindingSource(ds, "Personal");
private void button1_Click(object sender, EventArgs e)
{
    bs.MoveFirst();
}
```

35.6.2 Hinzufügen und Löschen

Dafür bietet die *BindingSource* die Methoden *Add, AddNew, Remove*, *RemoveAt*, *RemoveCurrent* und *RemoveFilter.*

BEISPIEL 35.9: Ein neuer Datensatz wird hinzugefügt

C#
```
bs.AddNew();
```

Der aktuelle Datensatz wird gelöscht:

```
bs.RemoveCurrent();
```

35.6.3 Aktualisieren und Abbrechen

Mit der *EndEdit*- bzw. *CancelEdit*-Methode der *BindingSource* kann der aktuelle Editiervorgang beendet bzw. abgebrochen werden.

BEISPIEL 35.10: Geänderte Daten vom *DataTable*-Objekt *dt* in die Datenbank übertragen.

C#
```
bs.EndEdit();
da.Update(dt);
```

HINWEIS: Wenn Sie die *EndEdit*-Methode nicht aufrufen, werden die geänderten Daten erst beim Weiterblättern in die *DataTable* übernommen.

35.6.4 Verwendung des BindingNavigators

Ein *BindingNavigator* eignet sich nur für die Zusammenarbeit mit einer *BindingSource*.

BEISPIEL 35.11: Ein *BindingNavigator* wird mit einem *BindingSource*-Objekt *bs* verknüpft.

C#
```
bindingNavigator1.BindingSource = bs;
```

Der *BindingNavigator* bietet alle Funktionen zum Weiterblättern, sowie zum Hinzufügen und zum Löschen – mit Ausnahme der "Speichern"- und der "Abbrechen"-Schaltfläche, die Sie selbst hinzufügen und implementieren müssen.

BEISPIEL 35.12: Ein *BindingNavigator*, dem Sie zwei Schaltflächen hinzugefügt haben, wird für das Speichern eines *DataTable*-Objekts *dt* und für das Abbrechen der aktuellen Operation "nachgerüstet".

C#

Speichern:

```
private void toolStripButton1_Click(object sender, EventArgs e)
{
    bs.EndEdit();
    da.Update(dt);
}
```

Abbrechen:

```
private void toolStripButton2_Click(object sender, EventArgs e)
{
    bs.CancelEdit();
}
```

Ergebnis

35.7 Die Anzeigedaten formatieren

Zum Formatieren der Inhalte manuell gebundener Steuerelemente ist etwas zusätzlicher Aufwand erforderlich. Die *Binding*-Objekte müssen separat erzeugt und mit Event-Handlern für das *Format*- und für das *Parse*-Event nachgerüstet werden.

BEISPIEL 35.13: Die Anzeige des Geburtsdatums wird formatiert.

C#

```
Binding b1 = new Binding("Text", bs, "Geburtstag");
```

Aufruf der Formatierungsmethoden (Implementierung siehe unten):

```
b1.Format += new ConvertEventHandler(this.DatToDateString);
b1.Parse +=  new ConvertEventHandler(this.DateStrToDat);
textBox3.DataBindings.Add(b1);
```

Datenquelle → Anzeige:

```
private void DateToDateString(object sender, ConvertEventArgs e)
{
 try
 { e.Value = Convert.ToDateTime(e.Value).ToString("d.M.yyyy"); }
  catch{}
}
```

Anzeige → Datenquelle:

```
private void DateStringToDate(object sender, ConvertEventArgs e)
{
   e.Value = Convert.ToDateTime(e.Value);
}
```

35.8 Praxisbeispiele

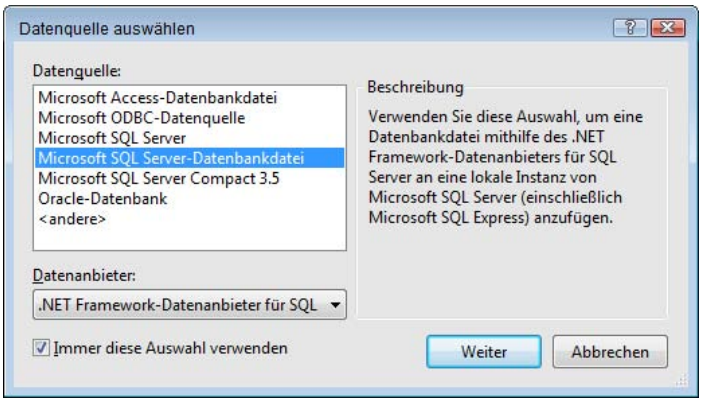
35.8.1 Einrichten und Verwenden einer Datenquelle

Im Zusammenhang mit dem Konzept der Datenquellen spielt der *TableAdapter* eine wichtige Rolle. Um Sinn und Zweck dieser assistentengestützten Technologie zu erkunden, wollen wir auf Basis einer Datenquelle ein Formular entwickeln, welches Informationen aus der *Customers*-Tabelle der *Northwind*-Datenbank des SQL Servers anzeigt.

Assistent zum Konfigurieren von Datenquellen

- Nachdem Sie ein neues Projekt vom Typ "Windows Forms-Anwendung" erzeugt haben, bringen Sie über das Menü *Daten/Datenquellen anzeigen* das Datenquellen-Fenster zur Anzeige.
- Oben links im Datenquellen-Fenster klicken Sie die Schaltfläche "Neue Datenquelle hinzufügen". Es startet der *Assistent zum Konfigurieren von Datenquellen*.
- Klicken Sie auf das "Datenbank"-Symbol und dann auf "Weiter".

- Im folgenden Dialogfenster wählen Sie die Schaltfläche "Neue Verbindung..." und es erscheint das Dialogfenster "Datenquelle auswählen".
- Hier klicken Sie auf den Eintrag "Microsoft SQL Server-Datenbankdatei". Genauso gut hätten Sie aber auch "Microsoft SQL Server" wählen können. Wir aber gehen diesmal davon aus, dass die Datenbank nicht auf dem SQL Server installiert ist, sondern als separate Datei *Northwind.mdf* zur Verfügung steht[1].

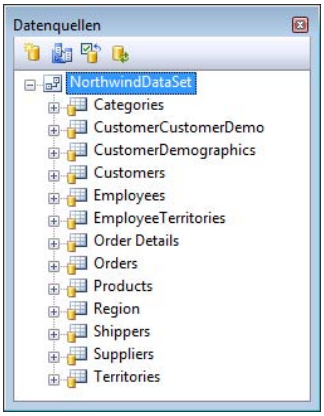


- Nach erfolgreichem Verbindungstest steht die Datenverbindung nun unter dem Namen *Northwind.mdf* zur Auswahl bereit.
- Anschließend werden Sie von einem Meldungsfenster befragt, ob Sie diese Datei in das Projekt kopieren wollen. Bestätigen Sie mit "Ja", denn Sie sparen sich damit eine Menge Ärger beim Pflegen der Anwendung bzw. bei deren späterer Weitergabe.
- Im nun folgenden Dialog können Sie guten Gewissens das Häkchen setzen, damit die Verbindungszeichenfolge als "NorthwindConnectionString" in der Anwendungskonfigurationsdatei gespeichert wird (ein Blick in die Datei *app.config* bestätigt Ihnen, dass der Eintrag in der *connectionStrings*-Sektion gelandet ist). Wenn Sie wollen, können Sie später per Code wie folgt darauf zugreifen:

```
...
 string connStr = WindowsApplication1.Properties.Settings.Default.NorthwindConnectionString;
...
```

- Schließlich offeriert Ihnen der Assistent, nachdem er die Datenbankinformationen abgerufen hat, das Dialogfenster "Datenbankobjekte auswählen". Hier können Sie die Tabellen, Ansichten, Gespeicherten Prozeduren oder Funktionen auswählen, die Sie für Ihre konkrete Anwendung brauchen.
- Die Datenquelle steht Ihnen jetzt im "Datenquellen"-Fenster zur freien Verfügung.

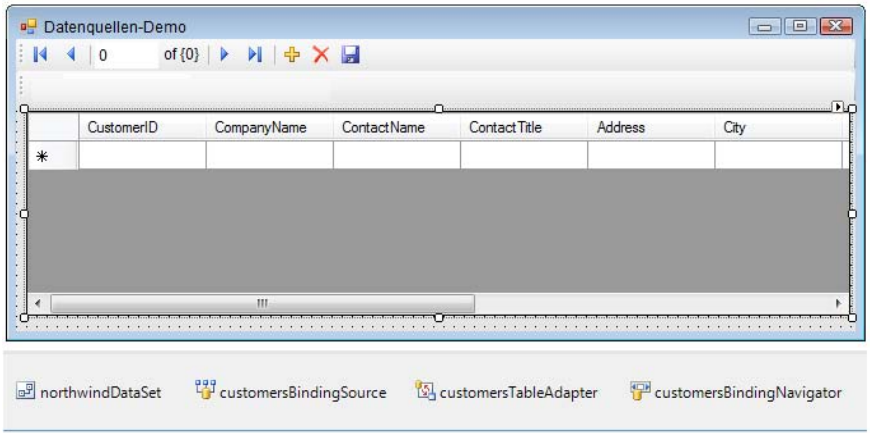
[1] Die ca. 3,7 MB große Datei *Northwind.mdf* finden Sie in den Buchbeispielen.



Verwenden der Datenquelle

Nachdem Sie per Drag & Drop die *Customers*-Tabelle vom Datenquellen-Fenster auf *Form1* gezogen haben geschehen wundersame Dinge: Wie von Geisterhand entfaltet sich ein *DataGridView*-Datengitter, dessen Spalten bereits beschriftet sind, auf dem Formular. Am oberen Rand hat ein *BindingNavigator* Platz genommen. Im vollen Komponentenfach tummeln sich folgende Objekte:

- *northwindDataSet*
  eine Instanz des typisierten DataSets
- *customersTableAdapter*
  ein typisierter *DataAdapter* für die *Customers*-Tabelle
- *customersBindingSource*
  die Datenanbindung des Formulars
- *customersBindingNavigator*
  navigiert *customersBindingSource*



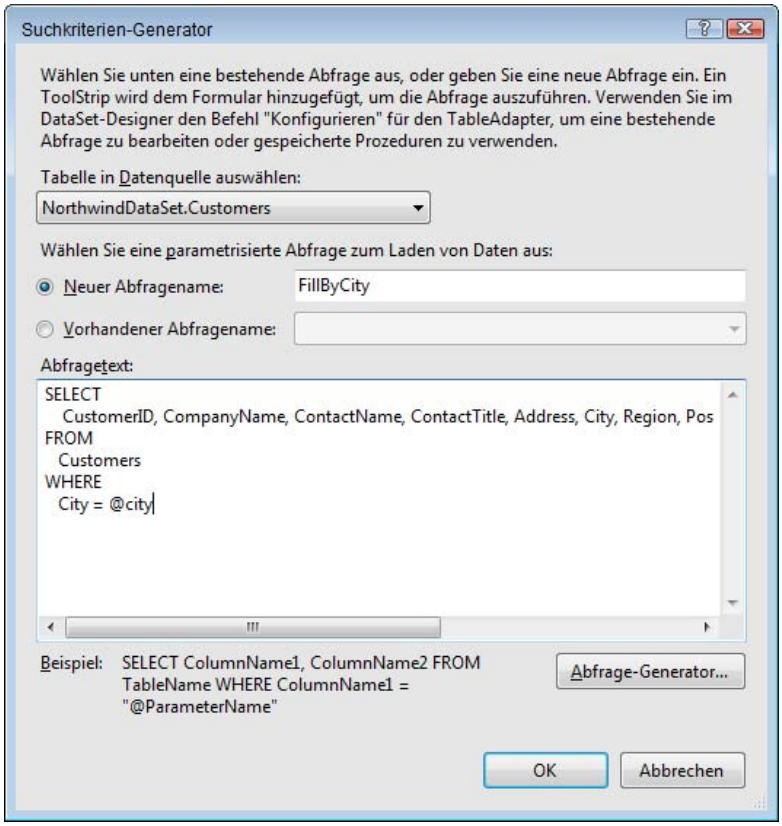
HINWEIS: Die Komponenten *northwindDataSet* und *customersTableAdapter* lassen sich im komfortablen DataSet-Designer weiter bearbeiten (Aufruf über Kontextmenü).

Test

Ohne dass Sie eine einzige Zeile Code geschrieben haben, liegt bereits eine voll funktionsfähige Anwendung vor, in der Sie nicht nur durch die Datensätze blättern können. Auch Editieren, Hinzufügen und (sofern die referenzielle Integrität nicht verletzt wird) Löschen sind möglich.

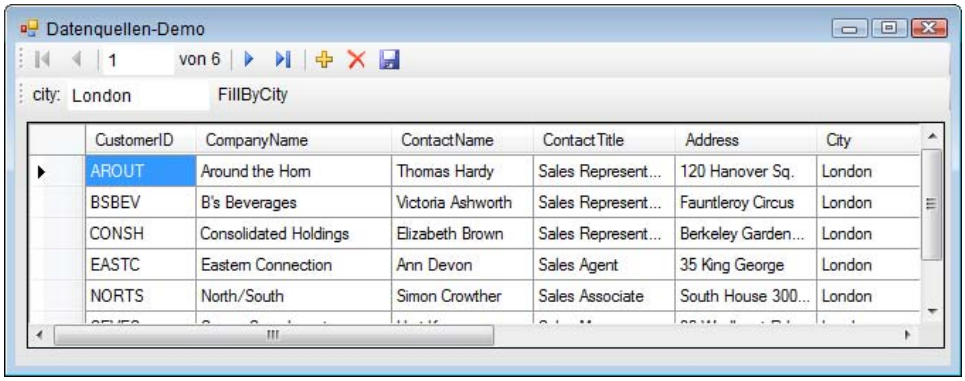
Abfragemethoden hinzufügen

Wir haben bis jetzt nur die Spitze des Eisbergs gesehen. Das Kontextmenü des *customersTableAdapter* bietet zum Beispiel auch einen Eintrag *Abfrage hinzufügen...*



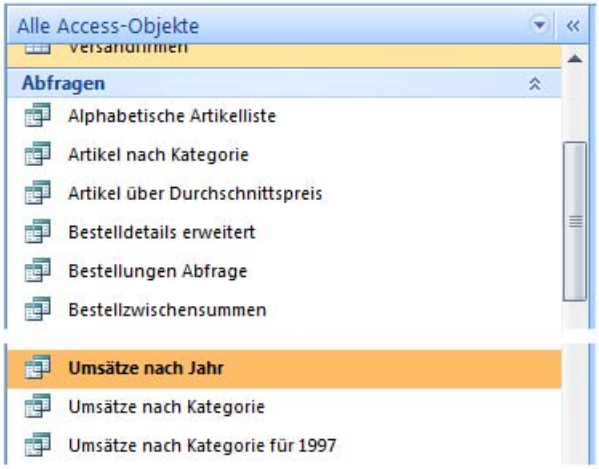
Im nachfolgenden Dialog vergeben Sie z.B. den Abfragenamen *FillByCity* und als Abfragetext den in obiger Abbildung angegebenen SQL-Befehl.

Nach dem OK wird abermals gezaubert: Unterhalb der Navigatorleiste erscheint ein automatisch generierter *ToolStrip* mit einer *TextBox* und einem *Button*. Nach Eingabe der gewünschten Stadt und Klick auf den Button "FillByCity" sehen Sie im *DataGridView* bereits das Abfrageergebnis.

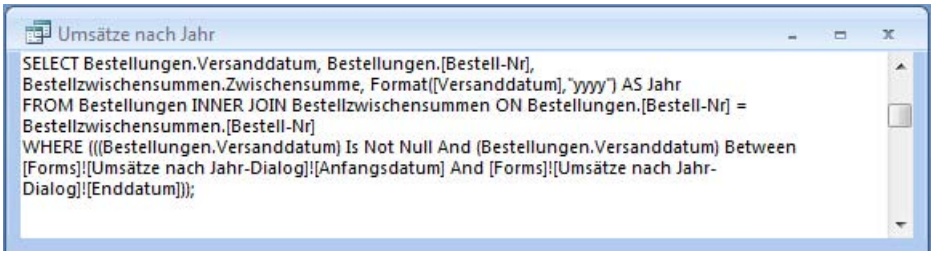


35.8.2 Eine Auswahlabfrage im DataGridView anzeigen

Die unter Microsoft Access gespeicherten Auswahlabfragen kann man quasi als Pendant zu den Stored Procedures des Microsoft SQL Servers betrachten. Öffnen Sie das Datenbankfenster von *Nordwind.mdb* und Sie sehen das zahlreiche Angebot an vorbereiteten Abfragen, die Sie natürlich auch selbst um weitere ergänzen können:



Hinter jeder Auswahlabfrage verbirgt sich in der Regel eine parametrisierte SQL-SELECT-Anweisung, die Sie sich im Access-Datenbankprogramm durch Öffnen der Entwurfsansicht über den Kontextmenübefehl *SQL-Ansicht* anschauen können. Dabei finden Sie auch die zu übergebenden Parameter und deren Datentypen leicht heraus:



Oberfläche

Ein *DataGridView*, zwei *TextBox*en und ein *Button* sollen für unseren Test genügen (siehe Laufzeitansicht am Schluss).

Quellcode

```
using System.Data.OleDb;
public partial class Form1 : Form
{
...
   private void button1_Click(object sender, System.EventArgs e)
   {
       string connStr = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;";
       OleDbConnection conn = new OleDbConnection(connStr);

       OleDbCommand cmd = new OleDbCommand("[Umsätze nach Jahr]", conn);
       cmd.CommandType = CommandType.StoredProcedure;
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
       OleDbParameter parm1 = new OleDbParameter("@Anfangsdatum", OleDbType.DBDate);
       parm1.Direction = ParameterDirection.Input;
       parm1.Value = Convert.ToDateTime(textBox1.Text);
       cmd.Parameters.Add(parm1);

       OleDbParameter parm2 = new OleDbParameter("@EndDatum", OleDbType.DBDate);
       parm2.Direction = ParameterDirection.Input;
       parm2.Value = Convert.ToDateTime(textBox2.Text);
       cmd.Parameters.Add(parm2);
```

Das *Command*-Objekt wird dem Konstruktor des *DataAdapter*s übergeben. Nach dem Öffnen der *Connection* wird die Abfrage ausgeführt. Die zurückgegebenen Datensätze werden in einer im *DataSet* neu angelegten Tabelle mit einem von uns frei bestimmten Namen *Jahresumsätze* gespeichert:

```
       OleDbDataAdapter da = new OleDbDataAdapter(cmd);
       DataSet ds = new DataSet();
       try
       {
```

```
            conn.Open();
            da.Fill(ds, "Jahresumsätze");
            conn.Close();
        }
        catch(Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }
```

Die Anzeige:

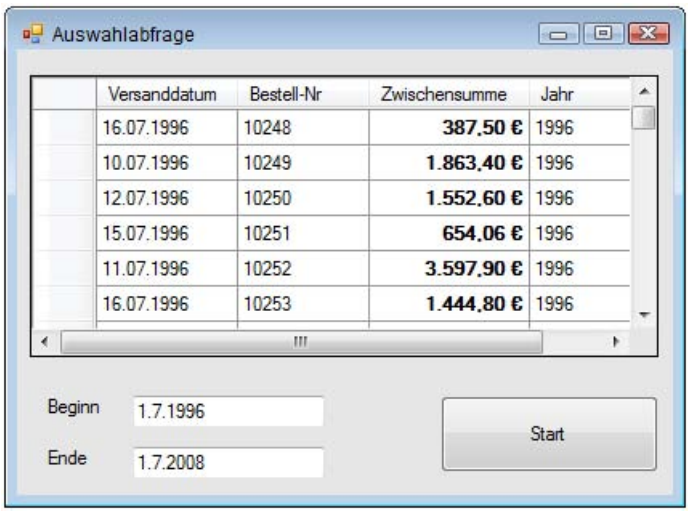
```
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "Jahresumsätze";
```

Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
        dataGridView1.Columns.Remove("Zwischensumme");
        DataGridViewTextBoxColumn tbc = new DataGridViewTextBoxColumn();
        tbc.DataPropertyName = "Zwischensumme";
        tbc.HeaderText = "Zwischensumme";
        tbc.Width = 80;
        tbc.DefaultCellStyle.Format = "c";
        tbc.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
        tbc.DefaultCellStyle.Font = new Font(dataGridView1.Font, FontStyle.Bold);
        tbc.DisplayIndex = 2;
        dataGridView1.Columns.Add(tbc);
    }
}
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



35.8.3 Master-Detailbeziehungen im DataGrid anzeigen

Das "gute alte" *DataGrid* kann mehrere Tabellen gleichzeitig verwalten, dies ist fast der einzige (wenn auch nicht unbedeutende) Vorteil gegenüber dem strahlenden Nachfolger *DataGridView*. Im vorliegenden Beispiel zeigen wir, wie man ohne viel Mehraufwand eine Darstellung von zwei verknüpften Tabellen (*Kunden* und *Bestellungen* aus der *Nordwind*-Datenbank) erreichen kann. Dabei lernen wir, wie man eine *DataRelation* erstellt und anwendet.

Oberfläche

Ein *DataGrid* und ein *Button* genügen für einen kleinen Test. Da bereits Visual Studio 2008 das *DataGrid* aus dem Werkzeugkasten vertrieben hat, müssen wir es aus der "Mottenkiste" wieder herausholen (Kontextmenü *Elemente auswählen...* und unter ".NET Framework-Komponenten" suchen).

Quellcode

```
using System.Data.OleDb;

public partial class Form1 : Form
{
   ...
```

Einrichten der Verbindung zur Datenbank:

```
    private void button1_Click(object sender, System.EventArgs e)
    {
        string connStr = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;" ;
        OleDbConnection conn = new OleDbConnection(connStr);
```

Die Tabelle *Kunden* wird in das *DataSet* geladen:

```
        string selStr = "SELECT KundenCode, Firma, Kontaktperson, Telefon FROM Kunden";
        OleDbDataAdapter da = new OleDbDataAdapter(selStr, conn);
        DataSet ds = new DataSet();
        conn.Open();
        da.Fill(ds, "Kunden");
```

Die Tabelle *Bestellungen* wird geladen:

```
        selStr = "SELECT Bestellungen.BestellNr, Bestellungen.KundenCode," +
                 " Bestellungen.Bestelldatum, Bestellungen.Versanddatum" +
                 " FROM Kunden, Bestellungen WHERE (Kunden.KundenCode =
                                                    Bestellungen.KundenCode)";
        da = new OleDbDataAdapter(selStr, conn);
        da.Fill(ds, "Bestellungen");
        conn.Close();
```

Die *DataRelation* wird zum *DataSet* hinzugefügt:

```
        ds.Relations.Add("KundenBestellungen", ds.Tables["Kunden"].Columns["KundenCode"],
                                               ds.Tables["Bestellungen"].Columns["KundenCode"]);
```

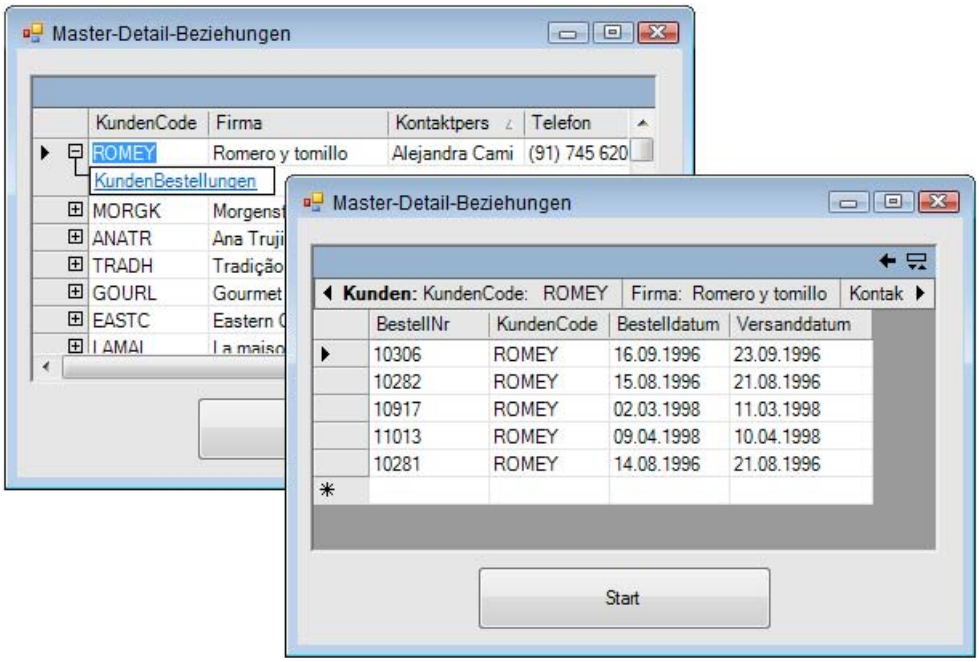
Anbinden des *DataGrid*:

```
        dataGrid1.SetDataBinding(ds, "Kunden");
    }
}
```

Test

Das *DataGrid* zeigt zunächst eine scheinbar normale Darstellung der Kundenliste. Nach dem Klick auf das Kreuzchen in der ersten Tabellenspalte können Sie die Darstellung expandieren. Nachdem Sie auf den Hotspot "KundenBestellungen" geklickt haben, erscheinen im *DataGrid* die gewünschten Detaildatensätze.

HINWEIS: Um zur Master-Tabelle zurückzukehren, klicken Sie auf den kleinen Pfeil rechts oben in der Titelleiste der Detailansicht.



35.8.4 Datenbindung Chart-Control

Selbstverständlich beherrscht auch das neue *Chart*-Control alle Möglichkeiten der Datenbindung, die Sorge, die Daten einzeln übergeben zu müssen, können Sie also gleich wieder vergessen.

HINWEIS: Grundsätzlich sollten Sie sich jedoch darüber im Klaren sein, welche Datenmengen Sie im Chart noch **effektiv** anzeigen können. Meist ist es sinnvoll, Daten zum Beispiel mit der bekannten SELECT TOP ...-Klausel zu filtern bzw. einzuschränken.

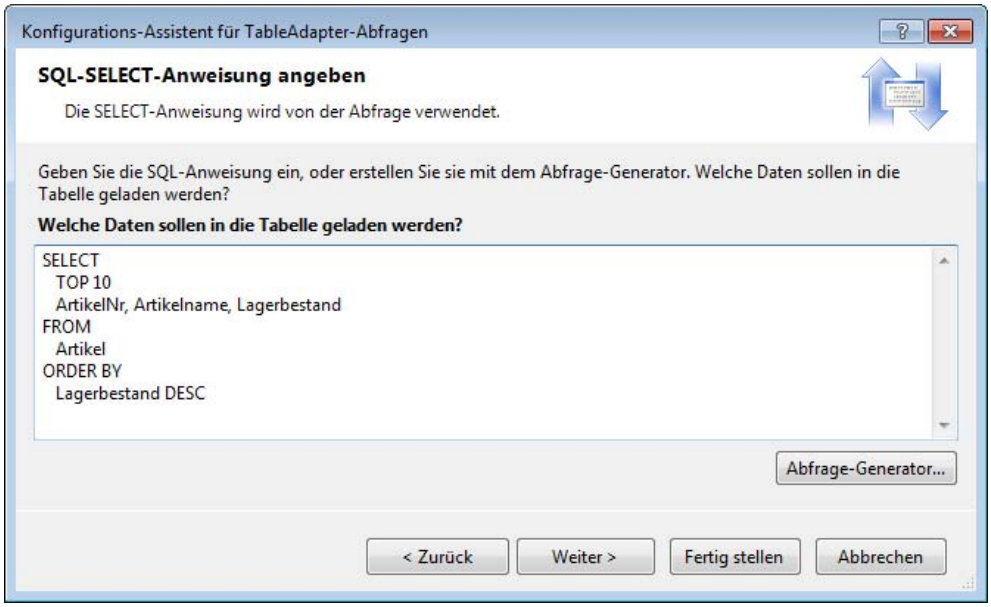
Unser kleines Beispielprogramm soll den Lagerbestand der Artikel aus der Access-Datenbank *Nordwind.mdb* in einem übersichtlichen Diagramm anzeigen. Allerdings werden Sie schnell feststellen, dass es kaum sinnvoll ist, mehrere hundert Artikel in einem Diagramm darzustellen. Aus diesem Grund beschränken wir uns auf die zehn häufigsten Artikel (Lagerbestand).

Oberfläche

Entwerfen Sie zunächst eine Windows Forms-Anwendung, der Sie zunächst die Datenbank *Nordwind.mdb* hinzufügen. Nachfolgend können Sie sich um das Erstellen einer neuen Datenquelle, basierend auf einem *DataSet*, kümmern.

HINWEIS: Für das Einbinden der Datenbank und das Erstellen des *DataSet*s sowie der übrigen datengebundenen Komponenten verweisen wir Sie an PB 35.8.1 "Einrichten und Verwenden einer Datenquelle".

Da wir uns auf die zehn häufigsten Artikel beschränken wollten, müssen wir entweder die bereits automatisch erzeugte *Fill*-Methode des erzeugten *TableAdapter*s (*ArtikelTableAdapter*) anpassen oder wir erstellen eine neue *Fill*-Methode (*FillbyTop10*) mit angepasstem SQL-Abfragestring (öffnen Sie dazu das neu erstellte Dataset):



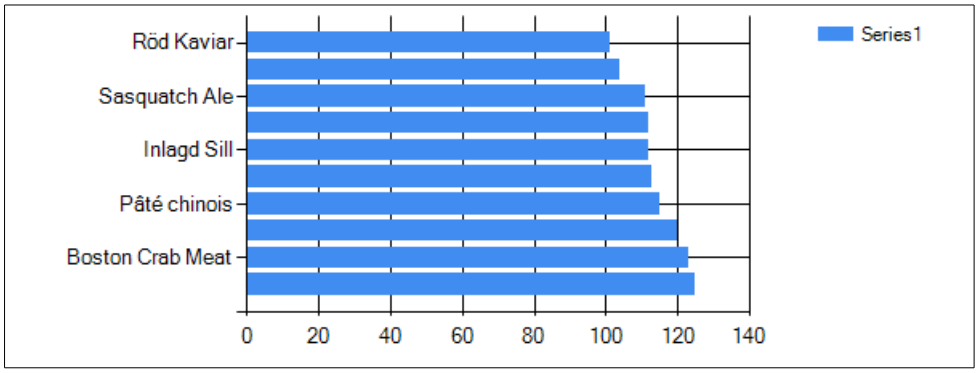
Mittels "TOP 10" beschränken wir uns auf zehn Datensätze, gleichzeitig beeinflussen wir mit "ORDER BY..." die Sortierfolge, um die zehn häufigsten Artikel abzurufen.

In *Form1* finden Sie jetzt neben dem *nordwindDataSet* auch noch die *artikelBindingSource* und den *artikelTableAdapter* vor. Fügen Sie abschließend ein *Chart*-Control hinzu, um dessen Konfiguration wir uns im Folgenden kümmern wollen.

Legen Sie zunächst die *DataSource*-Eigenschaft von *Chart1* auf *artikelBindingSource* fest. Nachfolgend können wir uns um die Eigenschaft *Series* kümmern, die für die Anzeige der einzelnen Datenreihen verantwortlich ist:

- Ändern Sie zunächst mittels Assistent den Diagrammtyp (*ChartType*). Wir wählen ein horizontales Balkendiagramm (*Bar*), so können später die Artikelbeschriftungen voll ausgeschrieben werden (Y-Achse).
- Über die Eigenschaften *XValueMember* und *YValueMember* können Sie die Tabellendaten der X- bzw. Y-Achse zuordnen. Doch Vorsicht: Beim vorliegenden Diagrammtyp müssen Sie die *Artikelnamen* der Eigenschaft *XValueMember* und den *Artikelbestand* der Eigenschaft *YValueMember* zuweisen.

Mit den bereits vorgenommenen Einstellungen würde zur Laufzeit folgendes Diagramm erzeugt werden:



Wie Sie sehen, werden nicht alle Y-Werte korrekt beschriftet, ansonsten sieht das Ergebnis doch schon recht ansprechend aus.

Um die ordentliche Beschriftung unserer zehn Artikel sicherzustellen, öffnen Sie zunächst den Assistenten für die *ChartAreas*-Eigenschaft. Suchen Sie hier die Eigenschaft *Axis* und öffnen Sie den nächsten Assistenten. In der Liste der Achsen wählen Sie "X axis" und legen für diese die Eigenschaft *Intervall* auf "1" fest. Damit werden jetzt alle Artikel ordentlich beschriftet.

Was jetzt noch bleibt, ist das optische Aufbessern des Diagramms:

- Den Diagramm-Titel können Sie über die *Titles*-Collection bestimmen. Fügen Sie mittels Assistent einen neuen Member hinzu und legen Sie dessen *Text*-Eigenschaft wie gewünscht fest (z.B. "Lagerbestand").
- Größe, Farbe, Position können Sie über die weiteren Eigenschaften des *Titles*-Members bestimmen.
- Die Legende (aktuell "Series 1") können Sie über die *Name*-Eigenschaft der jeweiligen Serie anpassen.

- Für mehr Eindruck sorgen Sie über die 3-D-Darstellung (Collection *ChartAreas*, *ChartArea1*, Eigenschaft *Enable3D*). Die Ansicht können Sie nachfolgend über *Rotation* und *PointDepth* noch wie gewünscht anpassen.
- Weitere optische "Spielereien" wollen wir an dieser Stelle nicht besprechen, wir wollen ja Ihre Fähigkeiten nicht unterschätzen :-).

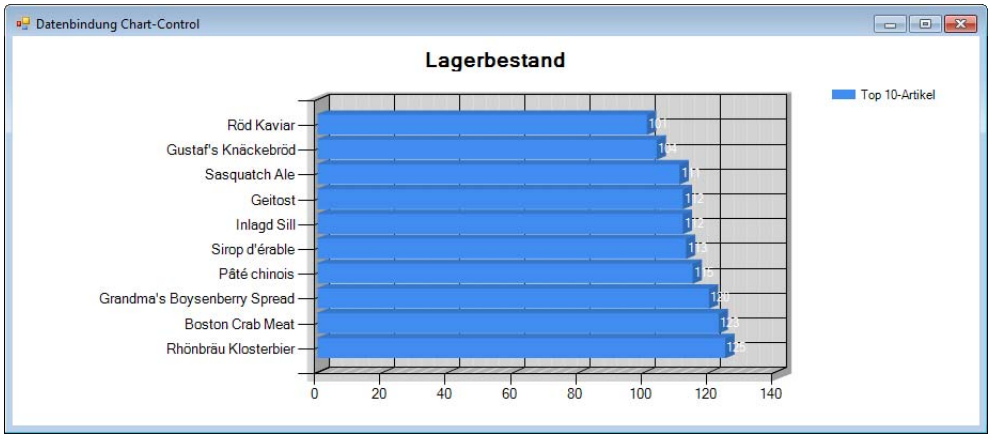
Quellcode

Der Datenquellen-Assistent hatte schon eine erste Quellcodezeile im *Form_Load*-Event generiert, mit der die Daten per *TableAdapter* geladen werden. Ändern Sie diese Anweisung und ersetzen Sie die *Fill*- mit der *FillByTop10*-Methode, damit auch nur 10 und nicht alle Datensätze geladen werden:

```
private void Form1_Load(object sender, EventArgs e)
{
    this.artikelTableAdapter.FillByTop10(this.nordwindDataSet.Artikel);
}
```

Test

Nach dem Start präsentiert sich Ihnen bereits das fertige Diagramm.



Kapitel 36

Erweiterte Grafikausgabe

Wie es die Kapitelüberschrift schon verrät, werden wir Sie im Folgenden mit den **erweiterten** Möglichkeiten der Grafikausgabe unter .NET traktieren. Grundkenntnisse der Grafikprogrammierung, siehe Kapitel 33, sind für die Lektüre dieses Kapitels unerlässlich.

Die wichtigsten Themen im Überblick:

- die GDI+-Transformationen
- Low-Level-Grafikzugriff
- erweiterte Techniken (Alpha-Blending, Animationen ...)
- 3D-Grafik unter .NET
- die Verwendung von GDI-Funktionen unter .NET

36.1 Transformieren mit der Matrix-Klasse

Die bereits im Abschnitt 33.3 über die Seitenkoordinaten vorgestellten Transformationen lassen sich auch mit Hilfe einer Transformationsmatrix realisieren, die Mathematiker unter den Lesern werden das zu schätzen wissen. Dass es nicht bei den beschriebenen Transformationen bleiben muss, und dass vieles auch einfacher realisierbar ist, zeigt der folgende Abschnitt.

36.1.1 Übersicht

Ausgangspunkt aller Überlegungen ist zunächst ist die Klasse *Matrix*, die vom Namespace *System.Drawing.Drawing2D* bereitgestellt wird. Hierbei handelt es sich intern um eine 3x3 Matrix, auch wenn Sie nur Zugriff auf die beiden ersten Spalten haben (die anderen Werte sind konstant).

$$\begin{vmatrix} m11 & m12 & 0 \\ m21 & m22 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Das Füllen der Matrix können Sie bereits mit Hilfe des Konstruktors bewerkstelligen, übergeben Sie einfach die Werte *m11* bis *dy*.

BEISPIEL 36.1: Erzeugen einer neuen Matrix

C#
```
Matrix m = new Matrix(1, 0,
                      0, 1,
                      0, 0);
```

HINWEIS: Zur besseren Übersicht haben wir im Listing noch Zeilenumbrüche hinzugefügt.

Alternativ können Sie über die *Elements*-Auflistung auf die einzelnen Elemente der Matrix zugreifen, es handelt sich jedoch **nicht** um ein zweidimensionales Array.

Die Elemente *dx* und *dy* stehen über die Eigenschaften *OffsetX* und *OffsetY* zur Verfügung.

36.1.2 Translation

Eine Verschiebung bzw. Translation realisieren Sie durch eine Änderung der *dy*-, *dy*-Werte.

$$x' = x + dx$$
$$y' = y + dy$$

... als Transformationsmatrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{vmatrix}$$

Alternativ können Sie die Methode *Translate* auf eine initialisierte Matrix anwenden. Übergeben Sie dazu die Werte für *dx* und *dy*.

36.1.3 Skalierung

Eine Größenänderung bzw. Skalierung erreichen Sie über die Werte *m11* bzw. *m22*:

$$x' = x \cdot m11$$
$$y' = y \cdot m22$$

... als Transformationsmatrix:

$$\begin{vmatrix} m11 & 0 & 0 \\ 0 & m22 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Gleiches erreichen Sie über die Methode *Scale*.

BEISPIEL 36.2: Verwendung von *Scale*

C#
```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Scale(2, 0);
```

36.1.4 Rotation

Auch an das Drehen des Koordinatensystems haben die .NET-Entwickler gedacht, allerdings genügt nicht das einfache Einsetzen des gewünschten Drehwinkels, sondern Sie müssen schon etwas rechnen:

$$x' = (x \cdot \cos\alpha) - (y \cdot \sin\alpha)$$
$$y' = (y \cdot \sin\alpha) - (y \cdot \cos\alpha)$$

... als Transformationsmatrix:

$$\begin{vmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Sie vermuten es sicher schon, mit Hilfe der Methode *Rotate* bietet sich ein wesentlich einfacherer Weg zum Drehen des Koordinatensystems.

BEISPIEL 36.3: Verwendung von *Rotate*

C#
```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Rotate(45);      // Gradangabe
```

36.1.5 Scherung

Eine Scherung in x- bzw. y-Richtung erreichen Sie über die folgende Transformationsmatrix:

$$x' = x + (m12 \cdot y)$$
$$y' = y + (m21 \cdot x)$$

... als Transformationsmatrix:

$$\begin{vmatrix} 1 & m12 & 0 \\ m21 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

BEISPIEL 36.4: Auch hier sind Sie mit der Methode *Shear* besser bedient:

C#
```
Matrix m = new Matrix(1, 0, 0, 1, 0, 0);
m.Shear(0.1f, 0);
```

36.1.6 Zuweisen der Matrix

Die Bearbeitung der Transformationsmatrix ist kein Selbstzweck, über die Eigenschaft *Transform* können Sie die mit obigen Methoden bearbeitete Transformationsmatrix einem *Graphics*-Objekt zuweisen. Nachfolgende Zeichenoperationen werden nun mit dieser Matrix verarbeitet.

BEISPIEL 36.5: Verschieben (20,10) und Skalieren (2,2) der Ausgabe

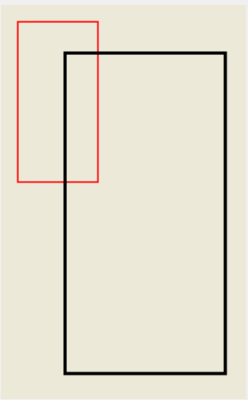
C#

```
Graphics g = this.CreateGraphics();
Matrix   m = new Matrix();
m.Translate(20, 10);
m.Scale(2, 2);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = m;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

oder bei direktem Bearbeiten der Matrix:

```
Graphics g = this.CreateGraphics();
Matrix m = new Matrix(2, 0, 0, 2, 20, 10);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
g.Transform = m;
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Ergebnis



36.2 Low-Level-Grafikmanipulationen

Möchten Sie Grafiken nicht nur anzeigen, sondern auch verändern, bieten sich auf den ersten Blick die *GetPixel*-/*SetPixel*-Methoden an. Diese ermöglichen den Zugriff auf die einzelnen Bildpunkte einer **Bitmap**[1], es ist demnach kein Problem, zum Beispiel eine Farbe durch eine andere auszutauschen.

[1] Dies ist wichtig, Sie müssen eine Pixelgrafik in das *Image* geladen haben.

Doch wer bereits erste Schritte auf diesem Gebiet unternommen hat, wird schnell enttäuscht sein. Das Auslesen und Setzen der einzelnen Punkte verbraucht bei komplexeren Aufgaben so viel Zeit, dass dies wohl kaum einem Programmnutzer zumutbar ist, es sei denn, man möchte ihn ärgern.

BEISPIEL 36.6: Drehen einer Bitmap[1] (in *picturebox1.Image*) um 90°

C#
```
private void button1_Click(object sender, EventArgs e)
{
   int x, y;
   Bitmap b1, b2;
   Color mypix;

   b1 = (Bitmap) pictureBox1.Image;
   b2 = new Bitmap(b1.Height, b1.Width);
   for(x = 0; x < b1.Width; x++)
      for (y = 0; y < b1.Height; y++)
      {
        mypix = b1.GetPixel(x, y);
        b2.SetPixel(b1.Height - y - 1, x, mypix);
      }
    pictureBox1.Image = b2;
    pictureBox1.Refresh();
}
```

Als Alternative bietet sich dem begeisterten GDI-Programmierer die Arbeit mit den *Device Independent Bitmaps* (DIB) an. Doch die Verwendung der entsprechenden Funktionen erfordert nicht nur eine genaue Kenntnis des GDI, sondern auch unnötigen Schreibaufwand, muss doch die DIB wieder in das Bitmap-Format zurück konvertiert werden. Vom zusätzlichen Speicherverbrauch wollen wir an dieser Stelle gar nicht erst sprechen.

Die Zauberworte für unsere Probleme heißen *LockBits* und *Scan0*. Dabei handelt es sich um Eigenschaften des *Bitmap*-Objekts. Mit *LockBits* wird eine Bitmap im Arbeitsspeicher für uns gesperrt, gleichzeitig können wir die Bitmap in einen für uns günstigen Datentyp umwandeln. Der eigentliche Clou ist *Scan0*, ein Pointer auf das erste Bitmap-Byte.

HINWEIS: Auch wenn es auf den ersten Blick so scheinen mag, *Scan0* ist kein direkter Ersatz für *GetPixel* und *SetPixel*. Wie die Bitmap-Daten aufgebaut sind, wie viele Spalten und Zeilen es gibt, bleibt unberücksichtigt. Ein Fehler bei der Arbeit mit diesem Pointer führt meist zu einem Programmabsturz.

36.2.1 Worauf zeigt Scan0?

Die allgemein Antwort lautet: auf das erste Byte der Bitmap. Wie die folgenden Daten aufgebaut sind und aus wie vielen Bytes sie bestehen, wird durch das Bitmap-Format bestimmt. Dieses können Sie mit *LockBits* direkt angeben:

[1] Ja, ja, es gibt dafür eine extra Methode, an dieser Stelle geht es aber nur um das Grundprinzip!

SYNTAX: `BitmapData LockBits(Rectangle rect, ImageLockMode flags, PixelFormat format);`

Übergeben werden der gewünschte Ausschnitt der Bitmap (im Zweifel alles), der Lockmode (meist *ReadWrite*) und das gewünschte Bitmap-Format.

C# unterstützt unter anderem folgende wichtige Bitmap-Formate:

Format	Beschreibung
Format1bppIndexed	Schwarz-Weiß-Bilder, bei denen jedes Pixel durch ein Bit dargestellt wird.
Format4bppIndexed	Bilder mit 16 Farben, ein Byte stellt somit die Informationen für zwei benachbarte Pixel zur Verfügung.
Format8bppIndexed	Bilder dieses Typs können 256 Farben darstellen. Damit entspricht ein Byte auch einem Pixel. Doch freuen Sie sich nicht zu früh, es handelt sich nicht um den direkten Farbwert, sondern nur um den Index in einer getrennt gespeicherten Farbpalette.
Format16bppRgb555, Format16bppRgb565	Bei diesem Format werden mit jeweils 5 bzw. 6 Bit für die drei Grundfarben die Farbwerte dargestellt bzw. abgespeichert. Das interne Format: pf15bit:0rrrrrgggggbbbbb pf16bit: rrrrrgggggbbbbb Dass die Arbeit mit derart verschachtelten Daten nicht unbedingt einfach ist, dürfte schnell ersichtlich sein.
Format24bppRgb	Das Wunschformat jedes Grafikprogrammierers: Jeder Punkt wird mit drei Bytes (RGB) zu je 8 Bit dargestellt. Der Zugriff auf derartige Bitmaps ist relativ problemlos realisierbar, beachten Sie jedoch, dass die Bildzeilen immer auf Vielfache von vier aufgerundet werden.
Format32bppArgb	Noch etwas schneller lassen sich 32-Bit-Bilder bearbeiten. Dies wird durch die bessere Speicherausrichtung (4 Byte) erreicht. Das vierte Byte hat normalerweise keine Bedeutung, kann aber für Transparenzinformationen genutzt werden. Der zusätzlich benötigte Speicher spielt heute wohl kaum noch eine Rolle.

36.2.2 Anzahl der Spalten bestimmen

Eigentlich müsste die Frage anders gestellt werden, da meist nicht die Anzahl der Bildpunkte, sondern die Anzahl der nötigen Bytes von Bedeutung ist. Die Pixelanzahl können Sie mit *Bitmap.Width* bestimmen, die Byte-Anzahl berechnet sich aus den jeweiligen Bildformaten, wie sie im vorhergehenden Abschnitt vorgestellt wurden.

32-Bit-Bild, Bitmap.Width = 300

*4 * Bitmap.Width = 1200 Bytes/Zeile*

Doch Vorsicht: Ein 24-Bit-Bild mit einer Breite von 299 Pixeln hat nicht etwa

*3 * Bitmap.Width = 897 Bytes/Zeile*

sondern

*((3 * Bitmap.Width + 3) DIV 4) * 4 = 900 Bytes/Zeile*

HINWEIS: Die Bitmaps werden in jeder Zeile auf das Vielfache von 4 Bytes aufgefüllt!

Aus diesem Grund finden Sie auch eine zusätzliche Eigenschaft *Stride*, die uns die tatsächliche Länge einer Bitmap-Zeile in Pixeln liefert.

BEISPIEL 36.7: Bestimmen der Anzahl von Füll-Bytes (24-Bit-Bitmap)

C#
```
using System.Drawing.Imaging;
...
    Bitmap b;
    BitmapData bmpData;
    int lineoffs;

    b = (Bitmap) pictureBox1.Image;
    bmpData = b.LockBits(new Rectangle(0, 0, b.Width, b.Height), ImageLockMode.ReadWrite,
                         PixelFormat.Format24bppRgb);
    lineoffs = bmpData.Stride - b.Width * 3;
```

HINWEIS: Dieser Offset muss nach der Abarbeitung einer Zeile zum Pointer hinzu addiert werden, um wieder auf das erste Byte der folgenden Zeile zu zeigen.

36.2.3 Anzahl der Zeilen bestimmen

Diese Antwort ist schnell gegeben, über die Eigenschaft *Bitmap.Height* steht Ihnen direkt der Wert zur Verfügung.

36.2.4 Zugriff im Detail (erster Versuch)

Folgende Reihenfolge müssen Sie beim direkten Zugriff auf die einzelnen Bitmap-Bytes beachten:

- Bitmap mit *LockBits* sperren,
- mit *Scan0* einen Pointer auf das erste Byte ermitteln,
- über *Marshal.Read…* die gewünschten Bytes/Integers etc. lesen,
- über *Marshal.Write…* die gewünschten Bytes/Integers schreiben,
- die Bitmap mit *UnlockBits* freigeben und
- eventuell die zugehörige *PictureBox* mit *Refresh* aktualisieren.

Den wohl wichtigsten Punkt dürfen wir natürlich auch nicht vergessen:

HINWEIS: Auch wenn die Konstanten *Format24bppRgb* oder *Format32bppArgb* heißen, lassen Sie sich nicht ins Boxhorn jagen! Die Bytes liegen immer in der Reihenfolge Blau-Grün-Rot bzw. Blau-Grün-Rot-Alpha im Speicher!

BEISPIEL 36.8: Alle Pixel der Grafik sollen auf Schwarz gesetzt werden (24-Bit-Bitmap)

C#

```
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...

    private void Button4_Click(object sender, EventArgs e)
    {
      Bitmap b = PictureBox1.Image as Bitmap;
      int x, y, offset;
      BitmapData bmpData;
      Byte p;
      IntPtr ptr;
```

Sperren der Bitmap:

```
      bmpData = b.LockBits(new Rectangle(0, 0, b.Width, b.Height),
                           ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
```

Pointer ermitteln:

```
      ptr = bmpData.Scan0;
      offset = 0;
      for (y=0; y < b.Height -1; y++ )
      {
        for (x=0; x < b.Width *3 -1; x++ )
          {
           p = 0;
```

Schreiben in den gewünschten Speicherbereich (alle Farbwerte = 0):

```
           Marshal.WriteByte(ptr, offset, p);
```

Offset für *Marshal.Write* setzen:

```
           offset += 1;
          }
      }
```

Freigabe der Bitmap und *PictureBox* aktualisieren:

```
      b.UnlockBits(bmpData);
      PictureBox1.Refresh();
    }
```

Haben Sie sich zu einem Test hinreißen lassen, wird es Ihnen sicher nicht anders als den Autoren ergangen sein. Nach quälenden Sekunden ist endlich das Bild schwarz. Grafikprogrammierer der ersten Stunde (Visual Studio 2002/2003) werden sich jetzt verwundert die Augen reiben, war doch dort dasselbe Beispiel ausreichend schnell (im Millisekunden-Bereich).

Ursache sind die mittlerweile quälend langsamen Zugriffe per *Marshal*-Objekt.

36.2.5 Zugriff im Detail (zweiter Versuch)

Nach unserem desaströsen ersten Versuch wollen wir es jetzt besser machen. Dazu bietet sich in C# die Verwendung von "unsicherem" Code an. Hier können wir mit Pointern schalten und walten wie wir wollen und natürlich auch reichlich Fehler produzieren.

HINWEIS: Gekennzeichnet werden unsichere Codeabschnitte mit dem *unsafe*-Bezeichner.

BEISPIEL 36.9: Wir versuchen uns mit dem gleichen Beispiel wie im vorherigen Abschnitt.

C#

```
int x, y;
BitmapData bmpData;

bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                       ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
unsafe
{
```

Hier ermitteln wir einen Byte-Pointer auf das erste Byte der Bitmap:

```
    byte* ptr = (byte*)bmpData.Scan0;
    int lineoffs = bmpData.Stride - bmp.Width * 3;
```

Für alle Zeilen und Spalten:

```
    for (y = 0; y < bmp.Height - 1; y++)
    {
        for (x = 0; x < bmp.Width * 3 - 1; x++)
        {
```

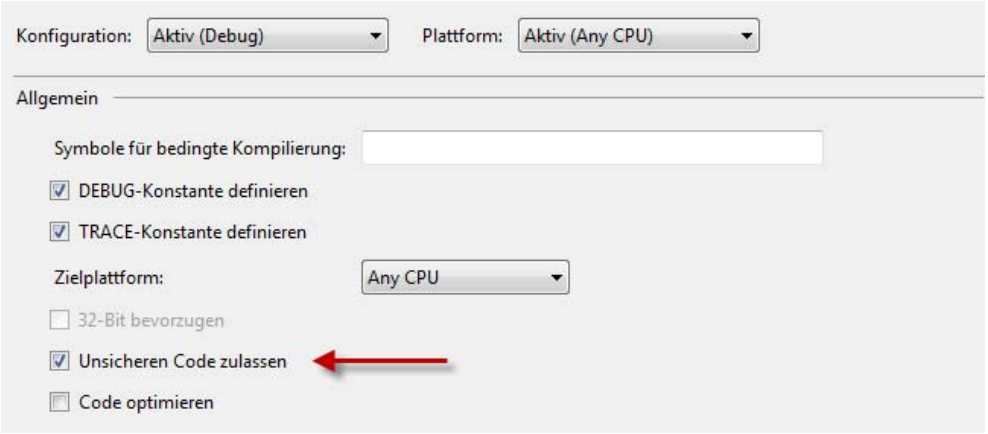
Wert setzen und Pointer inkrementieren:

```
            ptr[0] = 0;
            ptr++;
        }
```

Am Zeilenende den Offset addieren:

```
        ptr += lineoffs;
    }
}
bmp.UnlockBits(bmpData);
```

HINWEIS: Damit Sie das Beispiel kompilieren können, muss in den Projektoptionen auch das Kompilieren von unsicherem Code erlaubt werden.



Ein zweiter Test unseres optimierten Codes dürfte zwar schon etwas besser verlaufen, aber rund acht Sekunden sind trotzdem keine berauschende Zeit für die paar Pixel.

Weiter optimieren

Ein Blick auf unseren Code zeigt, dass wir eigentlich viel zu viele Berechnungen in den Schleifen ausführen (x, y berechnen, Offset addieren). Diesem Missstand wollen wir nun abhelfen. Um die unselige Offsetrechnerei zu vermeiden, erzeugen wir einfach eine 32-Bit-Grafik. Damit entfällt auch die Notwendigkeit, zwischen Zeilen und Spalten zu unterscheiden, wir können die geschachtelten Schleifen auflösen und der Pointer kann über den gesamten Speicherbereich iterieren:

```
BitmapData bmpData;
bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                       ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
unsafe
{
   byte* ptr = (byte*)bmpData.Scan0;
   int size = bmp.Height * bmp.Width * 4;  // 4 Bytes pro Pixel
   for (int i = 0; i < size; i++)
       ptr[i] = 0;
}
bmp.UnlockBits(bmpData);
```

Endlich dürfte die Ausführungsgeschwindigkeit auch den letzten Nörgler zufrieden stellen, 16 Millisekunden für die Beispielgrafik sind doch schon ganz gut – oder?

HINWEIS: Wer noch mehr mit der Zeit geizt, der kann auch einen Integer-Pointer inkrementieren (3 Zugriffe weniger pro Pixel).

Damit wird uns dieser Lösungsansatz auch für die weiteren Beispiele als Vorlage dienen.

BEISPIEL 36.10: Nur die Farbe Blau eines Pixels soll bearbeitet werden.

C#
```
using System.Drawing.Imaging;
using System.Runtime.InteropServices;
...
        public static void NurBlau(Bitmap bmp)
        {
            BitmapData bmpData;
            Byte blau;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                        ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = (bmp.Height) * (bmp.Width);
                for (int i = 0; i < size; i++)
                {
```

Achtung, der Blauwert ist immer das erste der vier Bytes:

```
                    blau = ptr[0];  // [1] = grün; [2] = rot
                    ptr[0] = 0;
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Das soll für einen ersten Eindruck genügen, die folgenden Beispiele zeigen an verschiedenen Aufgabenstellungen die Möglichkeiten, die Ihnen mit *Scan0* offen stehen.

HINWEIS: Ein Testprogramm für diese Funktionen finden Sie in den Beispieldaten zum Buch!

Im Folgenden wollen wir uns damit beschäftigen das Aussehen der Bilder, d.h. die einzelnen Farbwerte, zu verändern.

36.2.6 Invertieren

Eine der einfachsten Operationen ist das Invertieren einer Bitmap, die einzelnen RGB-Werte brauchen nur negiert zu werden, d h., der Farbwert ist von 255 abzuziehen.

BEISPIEL 36.11: Eine Bitmap wird invertiert.

C#
```
        public static void Invert(Bitmap bmp)
        {
            BitmapData bmpData;
```

BEISPIEL 36.11: Eine Bitmap wird invertiert.

```
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width * 4;
                for (int i = 0; i < size; i++)
```

HINWEIS: Wir verwenden *xor 255*, das ist noch mal etwas schneller als die Subtraktion.

```
                    ptr[i] = (byte)(ptr[i]^255);
                //    oder auch
                //    ptr[i] = (Byte)(255 - ptr[i]);
            }
            bmp.UnlockBits(bmpData);
        }
```

Ergebnis

36.2.7 In Graustufen umwandeln

Beim Umwandeln einer Farbgrafik in ein Graustufenbild werden die einzelnen Farben entsprechend ihrer Leuchtkraft bewertet und daraus ein Graustufenwert (8 Bit) berechnet. Dieser Wert wird nachfolgend allen drei Farbkanälen zugewiesen.

BEISPIEL 36.12: Eine farbige Bitmap wird in ein Graustufenbild transformiert.

C#

```
        public static void Grey(Bitmap bmp)
        {
            BitmapData bmpData;

            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                   ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
```

BEISPIEL 36.12: Eine farbige Bitmap wird in ein Graustufenbild transformiert.

```
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
                for (int i = 0; i < size; i++)
                {
```

Farbwerte auslesen:

```
                    Byte blau = ptr[0];
                    Byte grün = ptr[1];
                    Byte rot = ptr[2];
```

Grauwert berechnen:

```
                    Byte grau = (Byte) ((77 * blau + 151 * grün + 28 * rot) / 256);
```

Grauwert in die RGB-Bytes eintragen:

```
                    ptr[0] = ptr[1] = ptr[2] = grau;
```

Und Sprung auf das nächste Pixel:

```
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

Alternativ können Sie auch diese Anweisung verwenden:

```
ptr[0] = ptr[1] = ptr[2] = (Byte)((77 * ptr[0] + 151 * ptr[1] + 28 * ptr[2]) / 256);
```

36.2.8 Heller/Dunkler

Um ein Bild aufzuhellen oder dunkler zu machen genügt es, dass zu jedem Wert eine Konstante addiert wird. Um Werteüberläufe zu verhindern, müssten wir entweder bei jedem Wert abfragen, ob das Berechnungsergebnis den Wertebereich (255) überschreitet, oder wir legen gleich ein Array an, in dem für jeden der möglichen 256 Werte der neue Wert gespeichert ist. Insbesondere bei großen Bildern können Sie so wertvolle Sekunden sparen, da nur noch der Wert aus dem Array ausgelesen werden muss (LUT = Look-Up-Table).

BEISPIEL 36.13: Die Helligkeit einer Bitmap wird geändert.

C#

Normieren auf den Bereich 0... 255:

```
private static byte normiere(int Value)
    {
           if (Value < 0) return 0;
           if (Value > 255) return 255;
           return (byte)Value;
        }
```

BEISPIEL 36.13: Die Helligkeit einer Bitmap wird geändert.

Die Konvertierungsfunktion (*Value* = Änderung der Helligkeit):

```
public static void Brightness(Bitmap bmp, short Value)
{
```

Zunächst die LUT berechnen:

```
    Byte[] ar = new Byte[256];
    for (int i = 0; i < 256; i++)
        ar[i] = normiere(i + Value);
    BitmapData bmpData;
    bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                           ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
    unsafe
    {
        byte* ptr = (byte*)bmpData.Scan0;
        int size = bmp.Height * bmp.Width;
        for (int i = 0; i < size; i++)
        {
```

Hier lesen wir einfach die Werte aus der LUT aus:

```
            ptr[0] = ar[ptr[0]];
            ptr[1] = ar[ptr[1]];
            ptr[2] = ar[ptr[2]];
            ptr += 4;
        }
    }
    bmp.UnlockBits(bmpData);
}
```

Ergebnis

Übergeben Sie der Funktion einen positiven oder negativen Wert, um das Bild aufzuhellen oder abzudunkeln.

36.2.9 Kontrast

Um den Kontrast eines Bildes zu erhöhen, normieren wir zunächst die Farbwerte, indem wir diese in einen Integerwert umwandeln und 128 abziehen. Den resultierenden Wert multiplizieren wir mit einem konstanten Faktor, nachfolgend wird die Normierung durch Addition von 128 wieder aufgehoben. Da wir die Gleitkomma-Operationen nicht für jeden Pixel ausführen möchten (Performance!), verwenden wir wieder ein Array (LUT), in welchem wir die Farbwerte vorberechnen.

BEISPIEL 36.14: Kontrast einer Bitmap verändern

C#

```
public static void Contrast(Bitmap bmp, Single Value)
{
    Byte[] ar = new Byte[256];
    Value = 1 + Value / 100;
```

Zunächst die LUT berechnen:

```
    for (int i = 0; i < 256; i++)
        ar[i] = normiere((int)((i - 128) * Value) + 128);

    BitmapData bmpData;
    bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                           ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
    unsafe
    {
        byte* ptr = (byte*) bmpData.Scan0;
        int size = bmp.Height * bmp.Width;
```

Werte aus der LUT auslesen und zuweisen:

```
        for (int i = 0; i < size; i++)
        {
            ptr[0] = ar[ptr[0]]; ptr[1] = ar[ptr[1]]; ptr[2] = ar[ptr[2]];
            ptr += 4;
        }
    }
    bmp.UnlockBits(bmpData);
}
```

Ergebnis

36.2.10 Gamma-Wert

Möchten Sie den Gamma-Wert eines Bildes anpassen, ist die Rechnerei schon etwas aufwändiger. Auch hier hilft nur eine LUT weiter, sonst ist die Laufzeit nicht zu verantworten.

BEISPIEL 36.15: Der Gamma-Wert einer Bitmap wird angepasst.

C#

```
        public static void Gamma(Bitmap bmp, double Value)
        {
            Byte[] ar = new Byte[256];
```

Zunächst die LUT berechnen:

```
            for (int i = 0; i < 256; i++)
                ar[i] = (byte)Math.Min(255, (int)((255.0 *
                          Math.Pow(i / 255.0, 1.0 / Value)) + 0.5));
            BitmapData bmpData;
            bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                     ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
            unsafe
            {
                byte* ptr = (byte*)bmpData.Scan0;
                int size = bmp.Height * bmp.Width;
```

Auf alle drei Farbebenen anwenden:

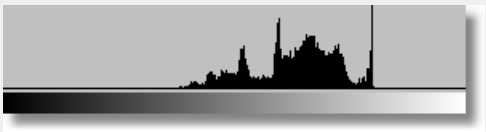
```
                for (int i = 0; i < size; i++)
                {
                    ptr[0] = ar[ptr[0]];
                    ptr[1] = ar[ptr[1]];
                    ptr[2] = ar[ptr[2]];
                    ptr += 4;
                }
            }
            bmp.UnlockBits(bmpData);
        }
```

36.2.11 Histogramm spreizen

Haben Sie Fotos mit falscher Belichtung aufgenommen, wird der verfügbare Dynamikumfang (0… 255) für die einzelnen Pixel nicht voll ausgeschöpft. Die Folge sind laue Farben, graue Bereiche, kein echtes Weiß und kein echtes Schwarz.

BEISPIEL 36.16: Bild mit zugehörigem Histogramm

Ergebnis



Wie Sie sehen ist links und rechts im Histogramm noch reichlich "Platz", der nicht genutzt wird.

HINWEIS: Durch das Spreizen des Histogramms wird der Dynamikumfang wieder erweitert, das "Grau in Grau" verschwindet.

BEISPIEL 36.17: Obiges Bild nach dem Spreizen des Histogramms

Ergebnis

BEISPIEL 36.18: Histogramm spreizen

C#

Die Umsetzung wird diesmal etwas aufwändiger:

```
public static void AutoAdjust(Bitmap bmp)
{
```

Unsere LUT wird für alle drei Farbebenen benötigt, deshalb die zweite Dimension:

```
    int[,] ar = new int[256, 3];
```

Beachten Sie auch den Wertebereich, diesen brauchen wir, wenn in der LUT zunächst das Histogramm abgespeichert wird.

BEISPIEL 36.18: Histogramm spreizen

```
BitmapData bmpData;
bmpData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                       ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
unsafe
{
    byte* ptr = (byte*)bmpData.Scan0;
    int size = bmp.Height * bmp.Width;
```

Erster Schritt ist das Erzeugen eines Histogramms, d.h., wir zählen die Häufigkeit bestimmter Farbwerte. Dazu nutzen wir gleich die LUT, die Größe stimmt ja überein:

```
    for (int i = 0; i < size; i++)
    {
        ar[ptr[0], 0]++;  // blau
        ar[ptr[1], 1]++;  // grün
        ar[ptr[2], 2]++;  // rot
        ptr += 4;
    }
```

Wir bearbeiten alle Farbkanäle

```
    for (int c = 0; c < 3; c++)
    {
```

Das Minimum (kleinster verwendeter Farbwert) im Histogramm suchen:

```
        int i = 0;
        while (ar[i, c] == 0) i++;
        int min = i;
```

Das Maximum (größter verwendeter Farbwert) suchen:

```
        i = 255;
        while (ar[i, c] == 0) i--;
        int max = i;
```

Die nötige Spreizung berechnen:

```
        Single scale = 255 / (Single)(max - min);
```

Die passende LUT für den Farbkanal ermitteln:

```
        for (int j = 0; j < 256; j++)
            ar[j, c] = normiere((int)((j - min) * scale));
    }
```

Zeiger auf das erste Byte setzen:

```
    ptr = (byte*)bmpData.Scan0;
```

Alle Farbkanäle mit der LUT verarbeiten:

```
    for (int i = 0; i < size; i++)
    {
```

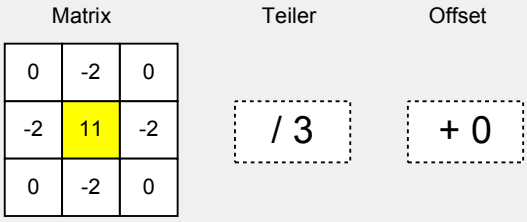
BEISPIEL 36.18: Histogramm spreizen

```
                ptr[0] = (byte)ar[ptr[0], 0];  // blau
                ptr[1] = (byte)ar[ptr[1], 1];  // grün
                ptr[2] = (byte)ar[ptr[2], 2];  // rot
                ptr += 4;
            }
        }
        bmp.UnlockBits(bmpData);
    }
```

36.2.12 Ein universeller Grafikfilter

Das Grundprinzip derartiger Verfahren ist die Verarbeitung der Farbwerte aus den umliegenden Pixeln als gewichtete Summe der Einzelwerte (Multiplizieren der Farbwerte mit dem jeweiligen Filtermatrixwert und nachfolgendes Addieren), das nachfolgende Teilen sowie das Addieren eines Offsets.

BEISPIEL 36.19: Ein Schärfefilter



Beispiel: Eine universelle Filter-Klasse entwickeln

Wie Sie der obigen Beschreibung bzw. der Skizze entnehmen können, brauchen wir für die einzelnen Grafikoperationen nicht jedes Mal "das Rad neu zu erfinden". Alle Berechnungen laufen immer nach dem gleichen Schema ab, eine universelle Klasse kann uns also die meiste Arbeit abnehmen.

Wir beginnen mit dem Einbinden der Namespaces:

```
...
using System.Text;
using System.Drawing;
using System.Drawing.Imaging;

namespace GDI_Filter
{
    public class GrafikFilter
    {
```

In dieser Eigenschaft speichern wir die Koeffizienten ab:

```
public int[ , ] Matrix = {{0,0,0},{0,0,0},{0,0,0}} ;
```

Der Teiler und der Offset:

```
public int Teiler = 0;
public int Offset = 0;
```

Eine schon bekannte Funktion zum Normieren von Bytewerten:

```
public byte normiere(int Value)
{
    if (Value < 0) return 0;
    if (Value > 255) return 255;
    return (byte)Value;
}
```

An die Methode *Execute* übergeben Sie später die zu bearbeitende Bitmap:

```
public void Execute(Bitmap bmp)
{
```

Wir benötigen zwischenzeitlich eine Kopie der Bitmap, da wir sonst mit schon bearbeiteten Farbwerten rechnen würden:

```
    Bitmap bmpQuelle = (Bitmap)bmp.Clone();
```

32-Bit-RGB-Bitmaps erzeugen:

```
    BitmapData dataQuelle = bmpQuelle.LockBits(new Rectangle(0, 0, bmp.Width,
                                    bmp.Height), ImageLockMode.ReadWrite,
                                    PixelFormat.Format32bppRgb);
    BitmapData dataZiel = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),
                                    ImageLockMode.ReadWrite,
                                    PixelFormat.Format32bppRgb);
```

Um nicht umständlich in mehreren Schleifendurchläufen die Matrixwerte abzufragen, erzeugen wir gleich drei Pointer für die jeweils abzufragenden Zeilen:

```
    unsafe
    {
        byte* pByte = (byte*)dataZiel.Scan0 + bmp.Width * 4 + 4;
        byte* pZeile0 = (byte*)dataQuelle.Scan0;
        byte* pZeile1 = (byte*)dataQuelle.Scan0 + bmp.Width * 4;
        byte* pZeile2 = (byte*)dataQuelle.Scan0 + bmp.Width * 8;
        int size = (bmp.Width - 2) * (bmp.Height - 2);
        for (int i = 0; i < size; i++)
        {
```

Was hier so kompliziert aussieht, ist nichts anders als das Wichten mit den Matrix-Werten, das Addieren, nachfolgende Teilen und das Zuweisen des Offsets für die drei Farbwerte:

```
            pByte[0] = normiere((((pZeile0[0] * Matrix[0, 0]) + (pZeile0[4] *
                            Matrix[1, 0]) + (pZeile0[8] * Matrix[2, 0]) +
```

```
                                    (pZeile1[0] * Matrix[0, 1]) + (pZeile1[4] *
                                    Matrix[1,1]) + (pZeile1[8] * Matrix[2, 1]) +
                                    (pZeile2[0] * Matrix[0, 2]) + (pZeile2[4] *
                                    Matrix[1,2]) + (pZeile2[8] * Matrix[2, 2])) / Teiler)
                                     + Offset);

                    pByte[1] = normiere((((pZeile0[1] * Matrix[0, 0]) + (pZeile0[5] *
                                    Matrix[1, 0]) + (pZeile0[9] * Matrix[2, 0]) +
                                    (pZeile1[1] * Matrix[0, 1]) + (pZeile1[5] *
                                    Matrix[1, 1]) + (pZeile1[9] * Matrix[2, 1]) +
                                    (pZeile2[1] * Matrix[0, 2]) + (pZeile2[5] *
                                    Matrix[1, 2]) + (pZeile2[9] * Matrix[2, 2]))
                                    / Teiler) + Offset);

                    pByte[2] = normiere((((pZeile0[2] * Matrix[0, 0]) + (pZeile0[6] *
                                    Matrix[1, 0]) + (pZeile0[10] * Matrix[2, 0]) +
                                    (pZeile1[2] * Matrix[0, 1]) + (pZeile1[6] *
                                     Matrix[1, 1]) + (pZeile1[10] * Matrix[2, 1]) +
                                    (pZeile2[2] * Matrix[0, 2]) + (pZeile2[6] *
                                    Matrix[1, 2]) + (pZeile2[10] * Matrix[2, 2]))
                                    / Teiler) + Offset);
```

Pointer um 4 Byte weiterbewegen (32 Bit RGB):

```
                    pByte += 4;
                    pZeile0 += 4;
                    pZeile1 += 4;
                    pZeile2 += 4;
                }
            }
            bmp.UnlockBits(dataZiel);
            bmpQuelle.UnlockBits(dataQuelle);
            bmpQuelle.Dispose();
        }
    }
}
```

HINWEIS: Wer mag kann sich an dieser Stelle ja versuchen, und eine Parallelisierung durchführen, was auf Mehrprozessorsystemen für einen weiteren Geschwindigkeitsschub sorgen dürfte.

In der Klasse *GrafikFilter* sind eigentlich alle Berechnungen gekapselt, was uns noch fehlt sind die Matrixwerte für die einzelnen Filter.

Die Klasse GrafikFilter im Einsatz

Am Beispiel von vier Filteralgorithmen wollen wir kurz die Verwendung der Klasse demonstrieren.

BEISPIEL 36.20: Bild schärfen

C#

```
public static void Sharpen(Bitmap bmp)
{
```

Klasse instanziieren:

```
    GrafikFilter gf = new GrafikFilter();
```

Werte festlegen:

```
    gf.Teiler = 3;
    gf.Matrix[0, 0] = 0; gf.Matrix[1, 0] = -2; gf.Matrix[2, 0] = 0;
    gf.Matrix[0, 1] = -2; gf.Matrix[1, 1] = 11; gf.Matrix[2, 1] = -2;
    gf.Matrix[0, 2] = 0; gf.Matrix[1, 2] = -2; gf.Matrix[2, 2] = 0;
```

Grafik bearbeiten:

```
    gf.Execute(bmp);
}
```

BEISPIEL 36.21: Weichzeichnen

C#

```
public static void Smoothing(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 8;
    gf.Matrix[0, 0] = 1; gf.Matrix[1, 0] = 1; gf.Matrix[2, 0] = 1;
    gf.Matrix[0, 1] = 1; gf.Matrix[1, 1] = 1; gf.Matrix[2, 1] = 1;
    gf.Matrix[0, 2] = 1; gf.Matrix[1, 2] = 1; gf.Matrix[2, 2] = 1;
    gf.Execute(bmp);
}
```

BEISPIEL 36.22: Emboss-Effekt

C#

```
public static void Emboss(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 1;
    gf.Offset = 127;
    gf.Matrix[0, 0] = -1; gf.Matrix[1, 0] = 0; gf.Matrix[2, 0] = -1;
    gf.Matrix[0, 1] = 0; gf.Matrix[1, 1] = 4; gf.Matrix[2, 1] = 0;
    gf.Matrix[0, 2] = -1; gf.Matrix[1, 2] = 0; gf.Matrix[2, 2] = -1;
    gf.Execute(bmp);
}
```

BEISPIEL 36.23: Gaußscher Weichzeichner

C#
```
public static void Gauss(Bitmap bmp)
{
    GrafikFilter gf = new GrafikFilter();
    gf.Teiler = 16;
    gf.Matrix[0, 0] = 1; gf.Matrix[1, 0] = 2; gf.Matrix[2, 0] = 1;
    gf.Matrix[0, 1] = 2; gf.Matrix[1, 1] = 4; gf.Matrix[2, 1] = 2;
    gf.Matrix[0, 2] = 1; gf.Matrix[1, 2] = 2; gf.Matrix[2, 2] = 1;
    gf.Execute(bmp);
}
```

HINWEIS: Von der Funktionsfähigkeit des Grafikfilters können Sie sich anhand des Beispielprogramms überzeugen!

36.3 Fortgeschrittene Techniken

Im Folgenden möchten wir noch auf einige Techniken eingehen, mit denen Sie die Ausgabeergebnisse bezüglich Geschwindigkeit und/oder Qualität verbessern können.

36.3.1 Flackerfrei dank Double Buffering

Sicher haben Sie auch schon vor dem Problem gestanden, dass Sie umfangreiche Grafiken ausgeben wollten, die Darstellung auf dem Bildschirm aber während dieser Zeit unerträglich flackert. Ein Beispiel zeigt die Problematik und im Anschluss auch den Lösungsansatz.

BEISPIEL 36.24: Ausgabe einiger Linien

C#
```
double von = System.Environment.TickCount;
Graphics g = this.CreateGraphics();
for (int i = 0; i < 800; i++)
{
    g.DrawLine(Pens.Red, 0, i, 400, i);
    g.DrawLine(Pens.Blue, 0, 0, 400, i);
    g.DrawLine(Pens.Green, 400, 0, 0, i);
}
g.Dispose();
double bis = System.Environment.TickCount;
label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Nach unerträglichen 0,45 Sekunden und viel Flackerei ist die Grafik endlich aufgebaut. Zwischenzeitlich sind auch Zeichenoperationen zu sehen, die wieder überdeckt werden.

Eine Puffer-Bitmap erzeugen

Mit Hilfe einer zusätzlichen Speicher-Bitmap können wir die Ausgaben zunächst unabhängig von der Oberfläche realisieren und nachfolgend ausgeben.

BEISPIEL 36.25: Fortführung des obigen Beispiels

C#

```
            double von = System.Environment.TickCount;
```

Bitmap erzeugen (entsprechend der Fenstergröße):

```
            Bitmap bmp = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
```

Grafikausgaben in der Bitmap vornehmen:

```
            Graphics g = Graphics.FromImage(bmp);
            for (int i = 0; i < 800; i++)
            {
                g.DrawLine(Pens.Red, 0, i, 400, i);
                g.DrawLine(Pens.Blue, 0, 0, 400, i);
                g.DrawLine(Pens.Green, 400, 0, 0, i);
            }
            g.Dispose();
```

Die Bitmap im Fenster wiedergeben:

```
            g = this.CreateGraphics();
            g.DrawImage(bmp, 0, 0);
            g.Dispose();
            bmp.Dispose();
            double bis = System.Environment.TickCount;
            label1.Text = ((bis - von) / 1000).ToString() + " s";
```

Mit 0,078 Sekunden Zeitbedarf können wir schon eine wesentliche Verbesserung feststellen, zusätzlich ist die Ausgabe flackerfrei und – last but not least – können wir die Bitmap mit jeder Paint-Operation ausgeben, ohne sie erneut aufbauen zu müssen.

Und was ist mit der PictureBox?

Hier haben wir es mit einem Sonderfall zu tun, die *PictureBox* beherrscht bereits "ab Werk" Double Buffering. Doch Achtung:

HINWEIS: Sie müssen mit der **enthaltenen** Grafik arbeiten, nicht mit der Bildschirmdarstellung.

Falsch:

```
Graphics g = pictureBox1.CreateGraphics();
...
```

Richtig:

```
Graphics g = Graphics.FromImage(pictureBox1.Image);
...
g.Dispose();
pictureBox1.Invalidate();
```

HINWEIS: Wichtig ist das *Invalidate*, sonst passiert auf dem Bildschirm nichts, bis die Grafik – zum Beispiel nach einem Verdecken – neu gezeichnet werden muss.

Sollten Sie keine Grafik in die *PictureBox* geladen haben, erzeugen Sie einfach eine entsprechende Grafik:

```
Bitmap bmp = new Bitmap(ClientRectangle.Width, ClientRectangle.Height);
pictureBox1.Image = bmp;
```

36.3.2 Animationen

Welcher Programmierer wird nicht ab und zu vom Spieltrieb übermannt? Auch die Autoren bilden hier keine Ausnahme. Zu jedem Spiel gehört auch etwas Action, und damit sind wir schon mitten im Thema angelangt. Wie können wir in .NET ein paar Bitmaps möglichst flackerfrei über den Bildschirm bewegen, ohne gleich auf DirectX zurückgreifen zu müssen?

Vorbereiten des Ausgabeobjekts

Im vorhergehenden Abschnitt war ja bereits die Rede von Double Buffering, eine Technik, die wir auch hier einsetzen wollen.

Die nahe liegende Lösung dürfte also das Erzeugen einer Hintergrundbitmap sein, auf der wir die einzelnen Bitmaps verschieben. Mit einem *Timer* blenden wir diese Bitmap zyklisch in den Vordergrund ein.

So weit so gut, das Flackern beim Bildaufbau können wir auf diese Weise vermeiden, allerdings macht uns Windows hier einen Strich durch die Rechnung. Es flackert trotzdem und zwar mit der Frequenz des Timers. Die Ursache findet sich in der Messagebehandlung für das Aktualisieren des Fensterhintergrunds (gilt auch für ein Control). Windows löscht bei jedem Refresh zunächst den Hintergrund mit der entsprechenden Hintergrundfarbe.

Lange Rede kurzer Sinn, mit Hilfe veränderter *ControlStyles* können wir Einfluss auf die Messagebehandlung durch Windows nehmen und stattdessen selbst für das Neuzeichnen des Controls/Fensters sorgen.

BEISPIEL 36.26: Ändern der Messagebehandlung im Formular-Konstruktor

C#

```
public Form1()
{
    InitializeComponent();
    SetStyle(ControlStyles.UserPaint, true);
    SetStyle(ControlStyles.AllPaintingInWmPaint, true);
    SetStyle(ControlStyles.DoubleBuffer, true);
}
```

Alternativ auch:

```
    SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
             ControlStyles.DoubleBuffer , true);
```

Die Änderungen im Einzelnen:

- *UserPaint* – Das Control zeichnet sich selbst, Windows bleibt außen vor.
- *AllPaintingInWmPaint* – Die Message WM_ERASEBKGND wird ignoriert.
- *DoubleBuffer* – Alle Zeichenoperationen werden zunächst im Hintergrund durchgeführt.

Nur alle drei Änderungen zugleich bewirken das gewünschte Ergebnis, nämlich kein Eingriff von Windows während unserer Zeichenaktivitäten.

Styles bei Controls setzen

Leider beschränkt sich dieses einfache Vorgehen zunächst auf das Formular, in allen anderen Controls sind die entsprechenden Styles nicht erreichbar (geschützte Methode).

Es hindert Sie aber nichts daran, einfach eine Ableitung der *Panel*-Komponente zu erzeugen, und hier die gewünschten Styles einzutragen.

BEISPIEL 36.27: Styles bei Controls setzen

C#

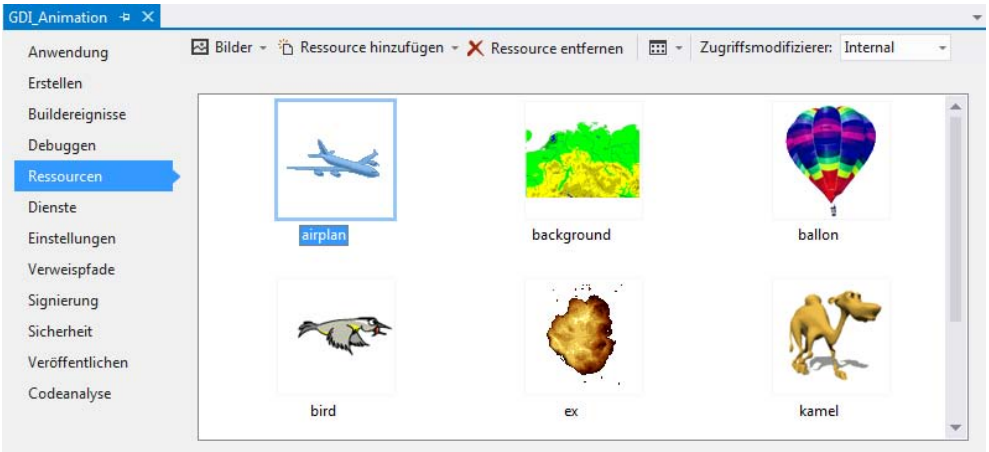
```
public partial class GrafikPanel : Panel
{
    public GrafikPanel()
    {
        InitializeComponent();
        SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
                 ControlStyles.DoubleBuffer, true);
    }

    public GrafikPanel(IContainer container)
    {
       ...
        SetStyle(ControlStyles.UserPaint | ControlStyles.AllPaintingInWmPaint |
                 ControlStyles.DoubleBuffer, true);
    }
}
```

Die eigentliche Grafikausgabe

Bisher hatten wir uns nur damit beschäftigt, wie das Fenster für die Ausgabe vorbereitet werden muss. Doch wie bringen wir unsere Grafik auf den Bildschirm?

Einige Ausschnitte aus dem Beispielprogramm zeigen die Vorgehensweise.

- Betten Sie zunächst die gewünschten Grafiken als Ressourcen in die Anwendung ein. Nutzen Sie dafür den Ressourcen-Editor von Visual Studio:



- Erzeugen Sie für alle benötigten Grafiken globale Variablen und laden Sie die Grafiken beim Initialisieren aus den Ressourcen nach:

```
public partial class Form1 : Form
{
    Bitmap bmp1 = GDI_Animation.Properties.Resources.bird;
    Bitmap bmp2 = GDI_Animation.Properties.Resources.ballon;
    Bitmap bmp3 = GDI_Animation.Properties.Resources.airplan;
    Bitmap bckbmp = GDI_Animation.Properties.Resources.background;
 ...
```

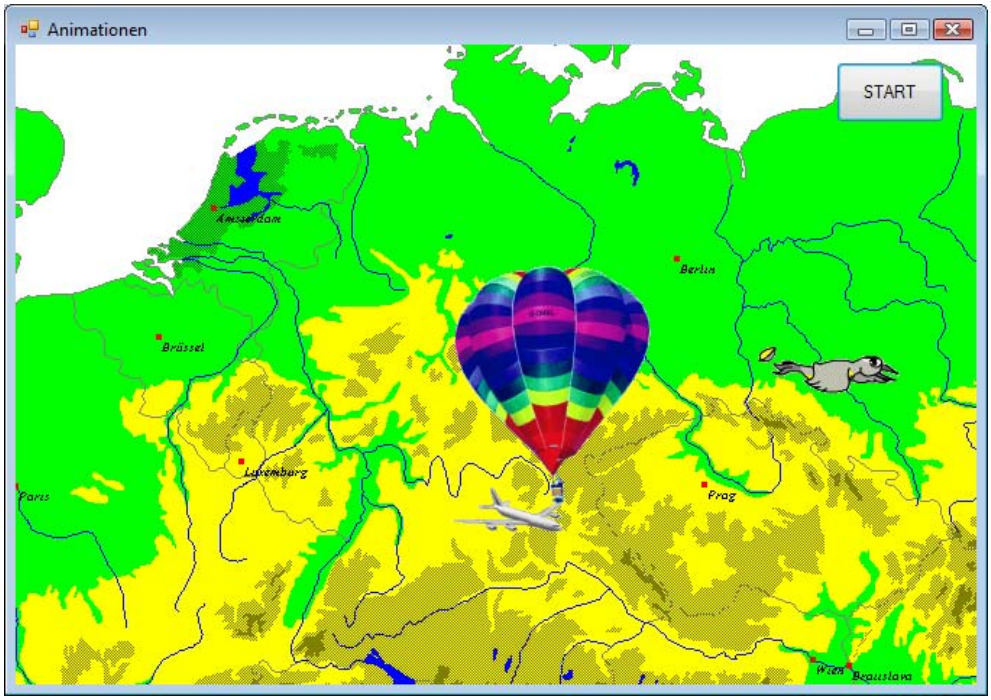
- Überschreiben Sie die *OnPaint*-Methode des Formulars und führen Sie hier Ihre Grafikoperationen aus:

```
    protected override void OnPaint(PaintEventArgs e)
    {
        e.Graphics.DrawImage(bckbmp, 0, 0);
        e.Graphics.DrawImage(bmp3, pos, pos);
        e.Graphics.DrawImage(bmp2, pos, 450 - pos);
        e.Graphics.DrawImage(bmp1, 2 * pos - 100, 200);
    }
```

- Nutzen Sie einen *Timer*, um zyklisch die *Invalidate*-Methode aufzurufen:

```
    private void timer1_Tick(object sender, EventArgs e)
    {
        pos++;
        if (pos > 500) pos = 0;
        this.Invalidate();
    }
```

Im Beispielprogramm verschieben wir drei Sprites über einer Hintergrundbitmap, ein Ruckeln werden Sie trotz des recht einfachen Verfahrens nicht feststellen:



Bemerkungen

- Beachten Sie, dass im Beispielprojekt für die Sprites GIF-Grafiken verwendet wurden. Diese ermöglichen es, Transparenz bereits in der Grafik festzulegen. Wir müssen die Objekte also nicht erst freistellen (Maskieren).
- Die Animation des Vogels, d h. dessen Flügelbewegung, steht im Mittelpunkt des folgenden Abschnitts.

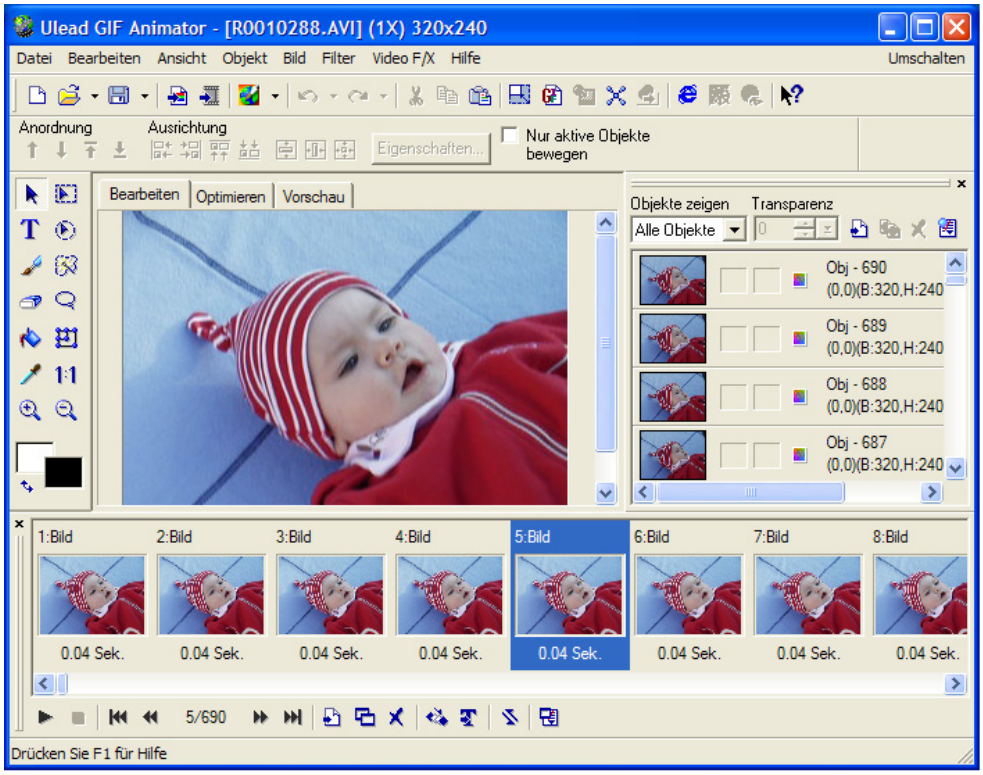
36.3.3 Animated GIFs

Soll der Eindruck von Bewegung entstehen, genügt es meist nicht (wie im vorhergehenden Abschnitt beschrieben) ein Sprite einfach über den Bildschirm zu schieben. Das funktioniert zwar ganz gut, ein echtes "Kino-Feeling" wird so aber nicht aufkommen. Besser funktioniert es mit Hilfe von animierten GIF-Grafiken (das gute alte "Daumenkino" lässt grüßen).

BEISPIEL 36.28: Einzelbilder aus einer animierten GIF-Datei

Wie Sie sehen, wird hier das Prinzip des Filmstreifens auf recht einfache Weise imitiert, nur die Anzahl der Bilder pro Sekunde ist wesentlich geringer, was aber in den meisten Fällen vollkommen genügt.

HINWEIS: Derartige Grafiken finden Sie zu Tausenden im Internet, oder Sie investieren selbst etwas Arbeit und erzeugen diese aus Videos oder Einzelgrafiken (zum Beispiel mit Hilfe des *Ulead GIF-Animator*s).



Wie lernen die Bilder das Laufen?

Auch für diese Aufgabe stellt das .NET-Framework die nötige Infrastruktur in Gestalt der *ImageAnimator*-Klasse zur Verfügung. Diese hat bereits einen integrierten Timer, der automatisch zwischen den einzelnen Bildern umschaltet.

BEISPIEL 36.29: Animierte GIFs wiedergeben

C# Fügen Sie die Animated GIFs als Ressourcen in Ihre Anwendung ein und laden Sie diese zur Laufzeit in eine Bitmap.

```
Bitmap bmp2 = GDI_AnimGIF.Properties.Resources.dino3;
Bitmap bmp1 = GDI_AnimGIF.Properties.Resources.dino6;
```

BEISPIEL 36.29: Animierte GIFs wiedergeben

Wer jetzt schon neugierig ist, wird leider enttäuscht. Die Grafik zeigt nur das erste Bild aus der GIF-Sequenz.

Melden Sie nachfolgend die Bitmap beim *ImageAnimator* an:

```
if (ImageAnimator.CanAnimate(bmp1))
    ImageAnimator.Animate(bmp1, this.OnNextFrame);
if (ImageAnimator.CanAnimate(bmp2))
    ImageAnimator.Animate(bmp2, this.OnNextFrame);
```

Verwenden Sie dazu die *Animate*-Methode, der Sie neben der jeweiligen Grafik auch einen Eventhandler übergeben können. Immer wenn ein neues Bild fällig ist, wird das Ereignis ausgelöst.

Erstellen Sie den Eventhandler, der für die Ausgabe verantwortlich ist:

```
private void OnNextFrame(object o, EventArgs e)
{
   this.Invalidate();
}
```

Geben Sie die Grafiken aus und schalten Sie mit *UpdateFrames* auf das jeweils nächste Bild in der Sequenz um:

```
protected override void OnPaint(PaintEventArgs e)
{
   e.Graphics.DrawImage(bmp2, 10, 10);
   e.Graphics.DrawImage(bmp1, 200, 10);
   ImageAnimator.UpdateFrames();
}
```

Ergebnis

36.3.4 Auf einzelne GIF-Frames zugreifen

Im vorhergehenden Abschnitt haben wir ja bereits eine Möglichkeit aufgezeigt, wie Sie als Programmierer die einzelnen Frames einer animierten GIF-Datei auslesen können, um zum Beispiel eine Animation zu realisieren. Allerdings haben Sie mit den o.g. Mitteln keinen Zugriff auf einen beliebigen Frame und Sie können auch nicht die Anzahl der Frames bestimmen.

Verantwortlich für diese Aufgaben ist ein *FrameDimension*-Objekt. Dem Konstruktor übergeben Sie die GUID der *FrameDimensionsList* des gewählten Bildes.

BEISPIEL 36.30: Abrufen eines FrameDimension-Objekts

C#
```
FrameDimension fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
```

HINWEIS: Über die *FrameDimensionsList* werden die einzelnen Frames bzw. verschiedenen Auflösungen des Bildes von .NET verwaltet.

Wiedergabe einzelner Frames

Möchten Sie einzelne Frames wiedergeben (zum Beispiel durch Verschieben eines *TrackBars*), brauchen Sie neben der Anzahl der Frames auch eine Möglichkeit, den aktuellen Frame zu setzen. In beiden Fällen hilft Ihnen das o.g. *FrameDimensions*-Objekt weiter.

BEISPIEL 36.31: Frameauswahl per *TrackBar*

C#
```
        Bitmap bmp = Properties.Resources.dino6;
        FrameDimension fdim;
...
        private void Form1_Load(object sender, EventArgs e)
        {
           fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
           trackBar1.Maximum = bmp.GetFrameCount(fdim)-1;
        }

        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            bmp.SelectActiveFrame(fdim, trackBar1.Value);
            Graphics g = this.CreateGraphics();
            g.Clear(this.BackColor);
            g.DrawImage(bmp,0,0);
        }
```

GetFrameCount liefert die Anzahl der verfügbaren Frames für die gewählte Bildabmessung, *SelectActiveFrame* setzt den aktiven Frame, der zum Beispiel beim Kopieren mittels *DrawImage* genutzt wird.

Erzeugen eines Bitmap-Strips

Sollen alle Bilder aus einer animierten Gif extrahiert und zum Beispiel als fortlaufender Streifen in einer Bitmap gesichert werden, können Sie sich am folgenden Code orientieren:

BEISPIEL 36.32: Extrahieren und Sichern aller Bilder einer animierten GIF-Bitmap

C#
```
        Bitmap bmp = Properties.Resources.dino6;
        FrameDimension fdim;
```

BEISPIEL 36.32: Extrahieren und Sichern aller Bilder einer animierten GIF-Bitmap

```
fdim = new FrameDimension(bmp.FrameDimensionsList[0]);
```

Hilfsbitmap erzeugen, mit der Breite=Frameanzahl*Framebreite:

```
Bitmap bmp2 = new Bitmap((bmp.GetFrameCount(fdim) - 1) * bmp.Width, bmp.Height);
Graphics g = Graphics.FromImage(bmp2);
```

Nacheinander die Frames kopieren:

```
for (int i = 0; i < bmp.GetFrameCount(fdim); i++)
{
    bmp.SelectActiveFrame(fdim, i);
    g.DrawImageUnscaled(bmp, i * bmp.Width, 0);
}
```

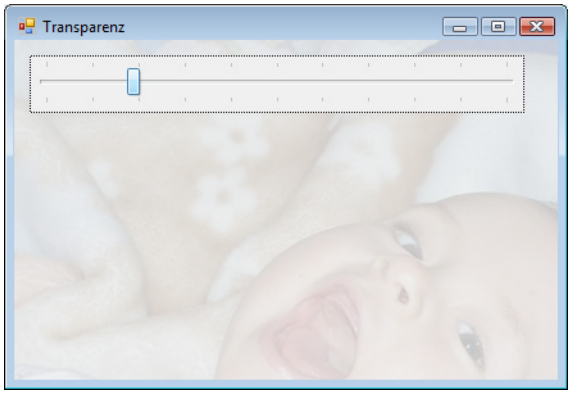
Die Hilfsbitmap können Sie gegebenenfalls einer *PictureBox* zuweisen:

```
pictureBox1.Image = bmp2;
g.Dispose();
```

Ergebnis

36.3.5 Transparenz realisieren

Sollen Bilder ein- bzw. ausgeblendet werden, müssen Sie das Rad nicht neu erfinden. Hier hilft Ihnen GDI+ mit seiner bereits eingebauten Fähigkeit, die Transparenz mit Hilfe des Alpha-Kanals zu steuern.



BEISPIEL 36.33: Ein Bild mit einer Transparenz zwischen 0 und 100% darstellen.

C#

```
private Bitmap bmp;
public Form1()
```

BEISPIEL 36.33: Ein Bild mit einer Transparenz zwischen 0 und 100% darstellen.

```
        {
            InitializeComponent();
```

Das Flackern unterbinden:

```
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
```

Bild aus den Ressourcen laden:

```
            bmp = GDI_Samples.Properties.Resources._0000003550;
        }
```

Bild zeichnen:

```
        protected override void OnPaint(PaintEventArgs e)
        {
```

Der Bitmap werden neue Attribute (eine *ColorMatrix*) zugewiesen:

```
            ImageAttributes iattr = new ImageAttributes();
            ColorMatrix m = new ColorMatrix();
```

Hier wird die Transparenz festgelegt:

```
            m.Matrix33 = trackBar1.Value / 100f;
            iattr.SetColorMatrix(m);
```

Leider ist die erforderliche Überladung der *DrawImage*-Methode etwas umfangreich:

```
            e.Graphics.DrawImage(bmp, new Rectangle(0, 0, bmp.Width, bmp.Height), 0, 0,
                                 bmp.Width, bmp.Height, GraphicsUnit.Pixel, iattr);
        }
```

Zu guter Letzt müssen wir noch die Bildaktualisierung erzwingen:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
           this.Invalidate();
        }
```

36.3.6 Eine Grafik maskieren

Für die Wiedergabe von Animationen benötigen Sie meist keine rechteckigen Objekte (Sprites) sondern bereits freigestellte Grafiken, d h., der Hintergrund ist transparent.

Zwei Verfahren bieten sich an:

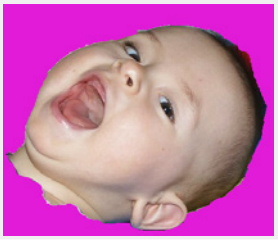
- Sie verwenden das GIF- oder PNG-Format und definieren bereits hier (im Zeichenprogramm) den transparenten Hintergrund. Lesen Sie eine derartige Datei mittels GDI+ ein, wird auch die Transparenz bei Ausgaben berücksichtigt.

- Sie füllen den Hintergrund mit einer definierten Farbe (z.B. Violett) und stellen die Grafik erste zur Laufzeit frei.

HINWEIS: Verwenden Sie ein verlustloses Komprimierverfahren (GIF, PNG, BMP), anderenfalls werden durch die Kompressionsartefakte die Masken unscharf wiedergegeben.

Variante 1 unterscheidet sich nicht von der normalen Wiedergabe mittels *DrawImage* etc., wir gehen deshalb nicht weiter darauf ein. Variante 2 erfordert etwas Vorarbeit.

BEISPIEL 36.34: Freigestellte Grafik, der Hintergrund wurde mit Violett gefüllt



Im Programm selbst müssen wird zunächst die Grafik laden und können dann mit Hilfe der Methode *MakeTransparent* den "Hintergrund" entfernen. Dazu lesen wir die Farbe eines der violetten Pixel aus:

```
Bitmap bmp1 = global::GDI_Maskieren.Properties.Resources._1;
Graphics g = this.CreateGraphics();
bmp1.MakeTransparent(bmp1.GetPixel(2, 2));
g.DrawImage(bmp1, 0, 0);
g.Dispose();
```

Nachfolgend lässt sich das Bild problemlos über einem anderen Hintergrund einblenden:

36.3.7 JPEG-Qualität beim Sichern bestimmen

Sicher hat es Sie auch schon gestört, dass Sie Ihre Bilder zwar problemlos im JPEG-Format sichern können, es aber zunächst keine Möglichkeit zu geben scheint, die Kompressionsrate einzustellen.

BEISPIEL 36.35: Sichern einer Grafik im JPEG-Format

C#

```
pictureBox1.Image.Save("Bild.jpg", ImageFormat.Jpeg);
```

Wer genauer hinschaut, der wird unter den zahlreichen Überladungen der *Save*-Methode einen Kandidaten für unsere Aufgabenstellung finden:

SYNTAX: `Image.Save (String, ImageCodecInfo, EncoderParameters)`

Doch woher nehmen wir *ImageCodecInfo* und *EncoderParameters*?

Dank der universellen Architektur ist die Verwendung nicht ganz trivial.

BEISPIEL 36.36: JPEG-Qualität beim Sichern bestimmen

C#

Zunächst müssen wir die *JPEG-Codec* aus der Liste der möglichen Codes ermitteln:

```
using System.Drawing.Imaging;
...
ImageCodecInfo myImageCodecInfo = null;
foreach (ImageCodecInfo codec in ImageCodecInfo.GetImageEncoders())
   if (codec.MimeType == "image/jpeg") myImageCodecInfo = codec;
```

Nachfolgend müssen wir noch einen Parameter erzeugen

```
EncoderParameters myParameter = new EncoderParameters();
```

und parametrieren (hier 25%):

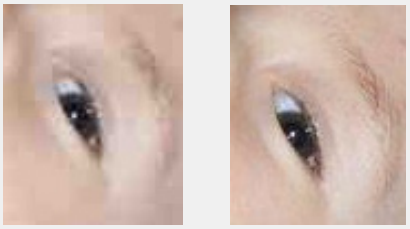
```
myParameter.Param[0] = new EncoderParameter(System.Drawing.Imaging.Encoder.Quality, 25);
```

Zu guter Letzt speichern wir mit Hilfe des Codes und des Parameters unsere JPEG-Grafik ab:

```
pictureBox1.Image.Save("Bild.jpg", myImageCodecInfo, myParameter);
```

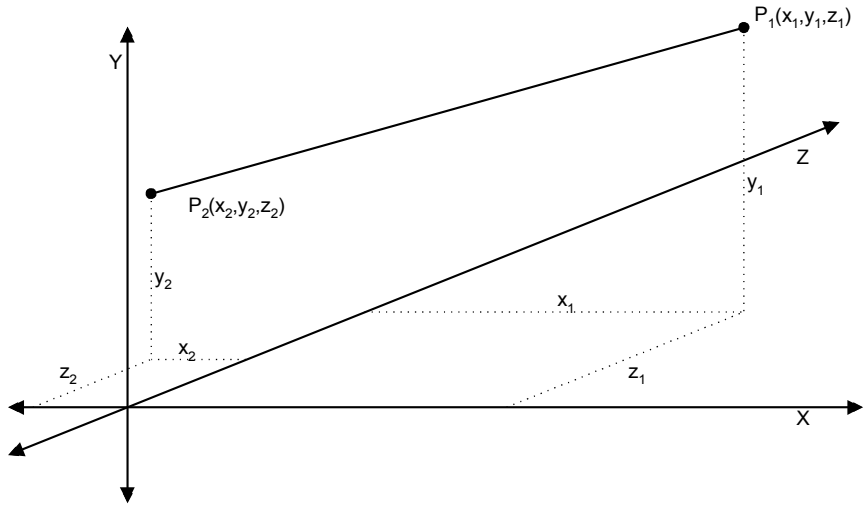
Ergebnis

Die folgenden Abbildungen zeigt einen Vergleich zwischen der Qualität 25% (deutliche Artefakte, kaum noch Details) und der Qualität von 75%.



36.4 Grundlagen der 3D-Vektorgrafik

Prinzipiell gelten für 3D-Vektorgrafiken auch die allgemeinen Ausführungen zur 2D-Grafik wie sie auch in GDI+ zum Einsatz kommt. Um aber bei der Menge der Koordinaten (jede Linie wird nicht durch vier, sondern durch sechs Werte repräsentiert) noch halbwegs den Überblick zu behalten, müssen wir das 2D-Konzept erweitern.



Zusätzlich zur x- und y-Koordinate müssen wir die z-Achse für die räumliche Tiefenwirkung verwenden.

Wir wollen im Folgenden davon ausgehen, dass der darzustellende Körper nur aus Linien besteht (so genanntes Drahtmodell), die Erweiterung der Algorithmen auf n-Ecke bzw. Kreise ist jederzeit möglich, würde jedoch den Rahmen dieses Kapitels sprengen. Ein Kreis muss durch seinen Mittelpunkt und mindestens einen Raumpunkt, der auf dem Umfang liegt, definiert sein. Übrigens lässt sich auch ein Kreis als Vieleck (Polygon) auffassen, die Rechenzeit verkürzt sich dadurch erheblich!

Ausgangspunkt für die weitere Arbeit ist die Suche nach einer sinnvollen Möglichkeit für das Speichern eines Körpers.

36.4.1 Datentypen für die Verwaltung

In C# bietet sich für diesen Zweck ein generisches *List*-Objekt an, das zum einen typsicher ist, zum anderen können beliebig viele Objekte gespeichert werden. Auch das spätere Abarbeiten der Liste ist mittels *foreach*-Schleife recht einfach realisierbar.

Grundlage des Körpers sind Linien, die durch Raumpunkte dargestellt werden. Jeder Raumpunkt besteht wiederum aus drei Koordinaten.

BEISPIEL 36.37: Die Umsetzung eines Raumpunkts in C#

C#

```
struct punkt3D
{
    public Double x, y, z;
    public PointF OnScreen;
}
```

In *OnScreen* speichern wir gleich die 2D-Ausgabekoordinaten für jeden 3D-Punkt, so bleibt alles zusammen.

Diese 3D-Punkte verpacken wir wiederum in universellen *Objects3D*-Objekten:

```
abstract class Objects3D
{
    public punkt3D[] points;
    public String Name;
    abstract public void Draw(Graphics g, Pen p);
}
```

HINWEIS: Bei der Klasse *Objects3D* handelt es sich um einen abstrakten Vorfahrtyp für unsere eigentlichen 3D-Objekte. Allen gemein ist ein Array von Punkten mit dem das 3D-Objekt beschrieben wird, ein Name (z.B. für das Debugging) sowie eine *Draw*-Methode, mit der sich das Objekt selbst zeichnet. Auf die Bedeutung der einzelnen Parameter kommen wir später zurück.

36.4.2 Eine universelle 3D-Grafik-Klasse

Die Liste zum Speichern eines 3D-Objekts verpacken wir in eine Klasse *c3D*:

```
class c3D
{
```

Zunächst allgemeine Optionen:

```
    public double Abstand = 1000;          // Betrachterabstand für die 2D-Darstellung
    public Pen Stift1 = Pens.Black;        // der Zeichenstift
```

Einige Hilfsvariablen und zwei kleine Arrays (LUT), in denen wir Werte vorberechnen:

```
    private double r =  Math.PI / 180;
    private Double[] sint = new Double[3];
    private Double[] cost = new Double[3];
```

Und hier die (generische) Liste der 3D-Objekte:

```
    private List<Objects3D> objectList = new List<Objects3D>();
```

Der Konstruktor:

```
    public c3D()
    {
        for (int i = -1; i < 2; i++)
```

```
        {
            sint[i + 1] = Math.Sin(i * r * 2);
            cost[i + 1] = Math.Cos(i * r * 2);
        }
    }
```

Neue Objekte fügen Sie in die Liste mit der Methode *AddObject* ein:

```
    public void AddObject(Objects3D o)
    {   objectList.Add(o);   }
```

Die weiteren Methoden der Klasse *c3D* (die eigentlichen Algorithmen stellen wir Ihnen erst in den folgenden Abschnitten vor):

```
    public void Scale(double f)     // Skalieren aller Objekte
    { ... }

    public void Translate(double x, double y, double z)  // Verschieben
    { ... }

    public void Rotate(int alpha, int beta, int gamma)   // Drehen
    { ... }
```

Hier nutzen wir die Vorteile der Vererbung, nach dem Zeichnen eines Koordinatensystems werden alle Objekte der Liste "aufgefordert", sich selbst per *Draw*-Methode zu zeichnen:

```
    public void Draw(Graphics g)
    {
        g.DrawLine(Pens.LightYellow, 200, 10, 200, 400);
        g.DrawLine(Pens.LightYellow, 10, 200, 400, 200);
        foreach (Objects3D o3d in objectList)
        {
```

Hier steht später die Umwandlung der 2D- in 3D-Koordinaten:

```
            // Erklärung folgt ...
            o3d.Draw(g, Stift1);
        }
    }
```

Übergeben wird lediglich das Ziel-*Graphics*-Objekt und der Zeichenstift.

36.4.3 Grundlegende Betrachtungen

Vor der weiteren Betrachtung stehen zwei Fragen:

- Wie sollen die Drehwinkel übergeben werden (relativ oder absolut)?
- Welche Darstellungsform auf dem Bildschirm soll gewählt werden?

Drehwinkel

Um den Aufwand möglichst niedrig zu halten, werden wir die Drehwinkel als relative Koordinaten bezüglich des letzten Drehpunkts interpretieren[1]. Dieses Vorgehen bietet mehrere Vorteile:

- Wir können die gleichen Algorithmen für die Rotation verwenden wie für die 2D-Grafik.
- Die Algorithmen sind nicht so komplex wie Algorithmen für die absolute Berechnung.
- Wir benötigen weniger Arbeitsspeicher.

Die letzte Aussage hört sich im Windows-Zeitalter etwas weltfremd an, um aber auch das Letzte an Geschwindigkeit aus unserem C#-Programm "herauszukitzeln", müssen wir statt der laufenden Berechnung von *Cos()* und *Sin()* auf vorbereitete Tabellen zugreifen. Da trigonometrische Funktionen nicht gerade superschnell sind, lässt sich auf diese Weise einiges an Rechenzeit einsparen.

Zurück zum Thema Speicherplatz: Für die absolute Positionierung auf 1/10° genau brauchen wir allein für die Sinus-Tabelle 360 * 10 Werte, und das im *double*-Format. Für die inkrementelle Methode sieht das ganz anders aus. Wir können einfach ein paar sinnvolle Werte speichern, die häufig gebraucht werden, z.B. ±1/10°, ±1°, ±10°, ±90° etc. Sie sehen, der Aufwand ist bedeutend geringer!

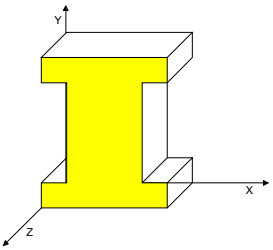
Wo viel Licht ist, da ist natürlich auch Schatten. Die inkrementelle Methode hat auch ihre Nachteile. Ein Problem ist beispielsweise, dass sich Berechnungsfehler immer weiter aufsummieren können. Nach genügend vielen Rotationen/Skalierungen wird es Ihnen kaum gelingen, den Ursprungskörper wiederherzustellen. Dem kann jedoch mit dem *double*-Format entgegengewirkt werden. Demgegenüber kommt die absolute Methode eventuell sogar mit Integer-Arithmetik aus. Wie gesagt, die Wahl der Methode sollten Sie von Ihrer jeweiligen Anwendung abhängig machen.

Bleibt noch das Problem der Darstellung auf dem Bildschirm.

Darstellungsmöglichkeiten

Für die Bildschirmdarstellung der 3D-Koordinaten können wir zwei verschiedene Verfahren verwenden:

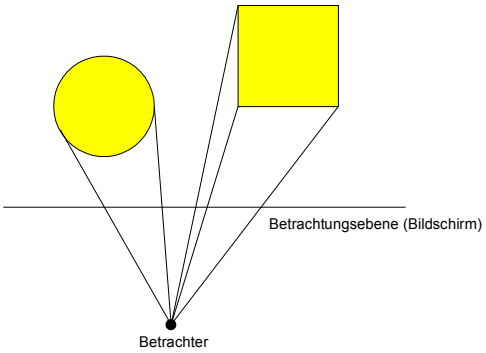
- **Parallelprojektion**
  Bei der Parallelprojektion, in diesem Fall der so genannten Kavalliersperspektive, bleibt die x-y-Ebene erhalten, z-Koordinaten werden in bestimmten Teilungsverhältnissen in der x-y-Ebene abgebildet.



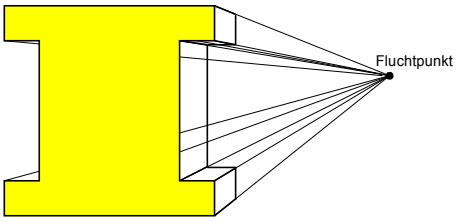
[1] Im Programm besteht die Möglichkeit, den Körper um jeweils ± 1 zu drehen.

Diese Darstellungsform verzerrt zwar den Körper, da aber die Strecken-Teilungsverhältnisse erhalten bleiben, wird die Parallelperspektive in der Praxis bevorzugt. Für ein Programm bedeutet das, dass x- und y-Koordinaten unverändert in die Betrachtung eingehen. Um die Darstellung nicht allzu sehr zu verzerren, sollten Sie den Neigungswinkel auf 18° reduzieren (Kavalliersperspektive 45°). Die z-Achse ist um 18° geneigt, zur x-Koordinate kommt also z/2 hinzu, zur y-Koordinate z/10.

- **Zentralprojektion** (Fluchtpunktperspektive**)**
  Die Zentralprojektion geht von einem Betrachterstandpunkt aus. Die Fluchtlinien Körperkante/ Betrachter werden in der Betrachtungsebene abgebildet:

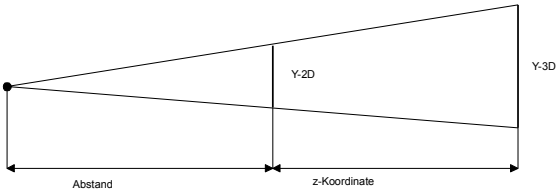


Durch diese Art der Darstellung entsteht ein Fluchtpunkt, der vom Betrachterstandpunkt abhängig ist.



Sie können diesen Effekt im Beispielprogramm nachvollziehen, indem Sie den Körper nach rechts/links verschieben und vergrößern/verkleinern.

Damit sind wir auch schon bei der Umsetzung angelangt. Für die Zentralprojektion benötigen wir zusätzlich den Abstand des Betrachters. Nach dem Strahlensatz gilt:



$$Y_{2D} = \frac{Y_{3D} \cdot Abstand}{Abstand + Z}$$

Entsprechendes ist natürlich auch für die x-Koordinate zutreffend.

HINWEIS: Bei ungünstigem Betrachterstandpunkt kann ein Laufzeitfehler "Division durch null" auftreten! Fangen Sie diesen ggf. mit *try-catch* ab.

Da in unserem Beispiel die Klasse *c3D* für die Konvertierung der 3D- in 2D-Koordinaten verantwortlich ist, können wir die Berechnung in die Methode *Draw* verlegen:

```
public void Draw(Graphics g)
{
    ...
    foreach (Objects3D o3d in objectList)
    {
       for (int i = 0; i < o3d.points.Length; i++)
           o3d.points[i].OnScreen = new PointF(Convert.ToSingle(o3d.points[i].x *
                                    Abstand / (Abstand + o3d.points[i].z) + links),
                                    Convert.ToSingle(-o3d.points[i].y * Abstand /
                                    (Abstand + o3d.points[i].z) + oben));
       o3d.Draw(g, Stift1);
    }
}
```

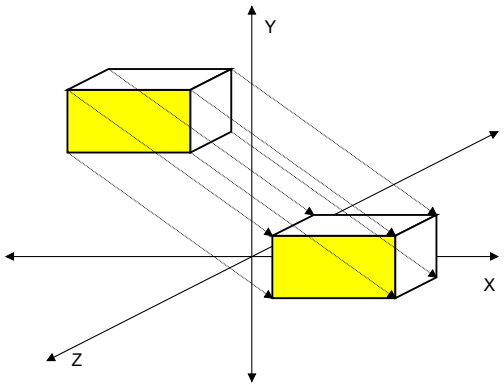
Wenden wir uns nun den eigentlichen Routinen für Translation, Streckung und Rotation zu.

36.4.4 Translation

Die Translation oder Verschiebung ist nichts anderes als die Addition von Punktvektor und Verschiebungsvektor. Was sich so kompliziert anhört, ist relativ trivial.

Jede Komponente des Punktvektors wird zu ihrem Verschiebungsvektor addiert.

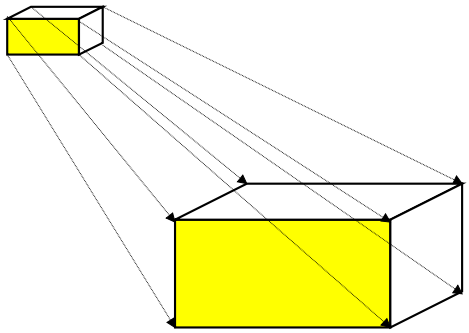
```
public void Translate(double x, double y, double z)
{
    foreach (Objects3D o3d in objectList)
    {
        for (int i = 0; i < o3d.points.Length; i++)
        {
            o3d.points[i].x += x;
            o3d.points[i].y += y;
            o3d.points[i].z += z;
        }
    }
}
```



HINWEIS: Um den gesamten Körper zu bewegen, müssen natürlich alle Punkte verschoben werden, wir kommen später darauf zurück.

36.4.5 Streckung/Skalierung

Neben der Verschiebung kann ein Körper auch vergrößert, d.h. skaliert werden. Die Skalierung wird durch Multiplikation des Punktvektors mit einer Konstanten (dem Skalierungsfaktor) realisiert.



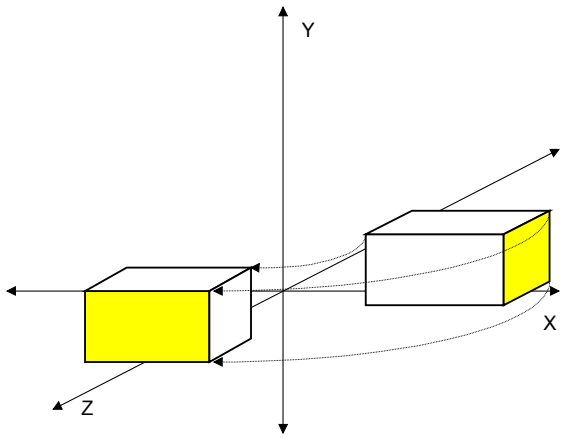
```
public void Scale(double f)
{
    foreach (Objects3D o3d in objectList)
    {
        for (int i = 0; i < o3d.points.Length; i++)
        {
           o3d.points[i].x *= f;
           o3d.points[i].y *= f;
           o3d.points[i].z *= f;
        }
    }
}
```

Je nachdem ob der Skalierungsfaktor größer oder kleiner 1 ist, wird der Körper vergrößert oder verkleinert.

36.4.6 Rotation

Die Programmierung eines Rotations-Algorithmus gestaltet sich naturgemäß schwieriger als für die Translation bzw. Skalierung. Um den Überblick nicht zu verlieren, führen wir folgende Notation ein:

- Rotation um die x-Achse α (alpha)
- Rotation um die y-Achse β (beta)
- Rotation um die z-Achse γ (gamma)



Für die Drehung um die einzelnen Achsen sind Transformationsmatrizen R_i gegeben:

$$R_X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

$$R_Y = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$R_Z = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Der neue Vektor bildet sich durch Multiplikation mit der entsprechenden Rotationsmatrix. Um unnötige Operationen zu vermeiden, werden die Matrizen aufgelöst und (soweit sinnvoll) berechnet (keine Multiplikationen mit null!).

Die Methode *Rotate* erwartet als Parameter die drei oben genannten Rotationswinkel. Im Programm sind nur Inkrementswinkel von -1°, 0°, +1° zulässig. Sie können die Anzahl der Winkelinkremente jedoch jederzeit in der *Init*-Prozedur ändern.

```
public void Rotate(int alpha, int beta, int gamma)
{
    punkt3D ph;
    if (alpha != 0)
    {
        alpha++;
        foreach (Objects3D o3d in objectList)
        {
            for (int i = 0; i < o3d.points.Length; i++)
            {
                ph = o3d.points[i];
                o3d.points[i].y = o3d.points[i].y * cost[alpha] -
                                  o3d.points[i].z * sint[alpha];
                o3d.points[i].z = ph.y * sint[alpha] + o3d.points[i].z * cost[alpha];
            }
        }
    }
    if (beta != 0)
    {
        beta++;
        foreach (Objects3D o3d in objectList)
        {
            for (int i = 0; i < o3d.points.Length; i++)
            {
                ph = o3d.points[i];
                o3d.points[i].z = o3d.points[i].z * cost[beta] -
                                  o3d.points[i].x * sint[beta];
                o3d.points[i].x = ph.z * sint[beta] + o3d.points[i].x * cost[beta];
            }
        }
    }
    if (gamma != 0)
    {
        gamma++;
        foreach (Objects3D o3d in objectList)
        {
            for (int i = 0; i < o3d.points.Length; i++)
            {
                ph = o3d.points[i];
                o3d.points[i].x = o3d.points[i].x * cost[gamma] -
                                  o3d.points[i].y * sint[gamma];
                o3d.points[i].y = ph.x * sint[gamma] + o3d.points[i].y * cost[gamma];
            }
        }
    }
```

HINWEIS: Die Rotation wird nur ausgeführt, wenn ein Inkrement ungleich null vorliegt. Auf diese Weise sollen unnötige Berechnungen vermieden werden.

36.4.7 Die eigentlichen Zeichenroutinen

An dieser Stelle werden Sie sicher enttäuscht sein. Wer hier noch irgendwelche aufwändigen Berechnungen sucht, hat die bisherigen Ausführungen nicht richtig gelesen. In den Elementen *punkt3D.OnScreen* befinden sich bereits vor dem Aufruf der Objekt-*Draw*-Methoden die 2D-Koordinaten, wie Sie für die GDI+-Ausgaberoutinen erforderlich sind. Deshalb reicht es jetzt, die Punktkoordinaten den GDI-Anweisungen zuzuweisen.

Ausgabe einer Linie

Definition einer neuen Klasse, abgeleitet von *Objects3D*:

```
class line3D : Objects3D
    {
```

Im Konstruktor erwarten wir die Ursprungskoordinaten:

```
        public line3D(double x1, double y1, double z1, double x2, double y2, double z2)
        {
```

Abspeichern in *points*:

```
            points = new punkt3D[2];
            points[0].x = x1; points[0].y = y1; points[0].z = z1;
            points[1].x = x2; points[1].y = y2; points[1].z = z2;
            Name = "line3D";
        }
```

Und hier die Zeichenroutine, die die abstrakte *Draw*-Methode von *Objects3D* ersetzt:

```
        override public void Draw(Graphics g, Pen p)
        {  g.DrawLine(p, points[0].OnScreen, points[1].OnScreen);  }
    }
```

Die Verwendung dürfte recht einfach sein, dank intensiver Vorarbeit durch die Klasse *c3D*.

Ausgabe eines Koordinatensystems

Auch hier leiten wir von *Objects3D* ab:

```
class coordinateSystem3D : Objects3D
{
```

Der Konstruktor erwartet nur die Achslänge, den Rest berechnen wir selbst:

```
        public coordinateSystem3D(double size)
        {
            points = new punkt3D[6];
```

X-Achse:

```
            points[0].x = -size;  points[0].y = 0;     points[0].z = 0;
            points[1].x = size;   points[1].y = 0;     points[1].z = 0;
```

Y-Achse:

```
            points[2].x = 0;      points[2].y = -size; points[2].z = 0;
            points[3].x = 0;      points[3].y = size;  points[3].z = 0;
```

Z-Achse:

```
            points[4].x = 0;      points[4].y = 0;     points[4].z = -size;
            points[5].x = 0;      points[5].y = 0;     points[5].z = size;
            Name = "coordinateSystem3D";
        }
```

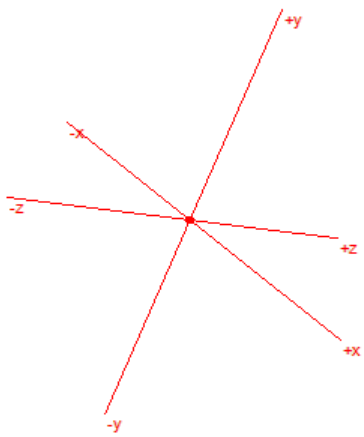
In der *Draw*-Methode müssen wir jetzt schon etwas mehr tippen:

```
        override public void Draw(Graphics g, Pen p)
        {
```

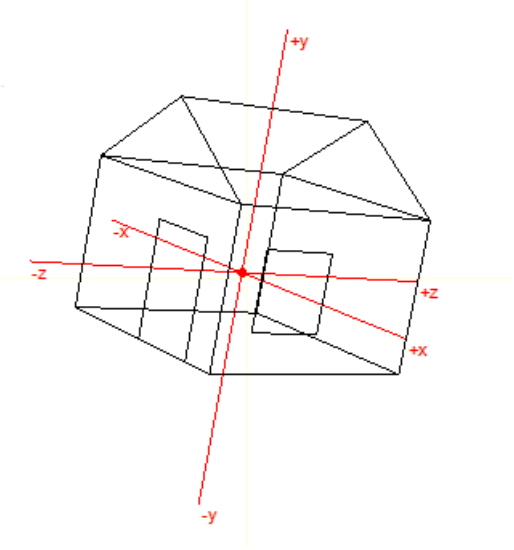
Ausgabe der Achsen und der Koordinatenangaben:

```
            g.DrawLine(Pens.Red, points[0].OnScreen, points[1].OnScreen);
            g.DrawString("+x", new Font("Arial", 8), Brushes.Red, points[1].OnScreen);
            g.DrawString("-x", new Font("Arial", 8), Brushes.Red, points[0].OnScreen);
            g.DrawLine(Pens.Red, points[2].OnScreen, points[3].OnScreen);
            g.DrawString("+y", new Font("Arial", 8), Brushes.Red, points[3].OnScreen);
            g.DrawString("-y", new Font("Arial", 8), Brushes.Red, points[2].OnScreen);
            g.DrawLine(Pens.Red, points[4].OnScreen, points[5].OnScreen);
            g.DrawString("+z", new Font("Arial", 8), Brushes.Red, points[5].OnScreen);
            g.DrawString("-z", new Font("Arial", 8), Brushes.Red, points[4].OnScreen);
        }
    }
```

Mit den obigen Anweisungen wird später das folgende Koordinatensystem gezeichnet:



Nach der Definition einiger Linien entsteht zum Beispiel das folgende 3D-Objekt:



HINWEIS: Das komplette 3D-Grafikbeispiel finden Sie im Praxisbeispiel 36.6.2.

36.5 Und doch wieder GDI-Funktionen ...

Wer geglaubt hätte, GDI+ sei so etwas wie der komplette Ersatz von GDI, sieht sich in vielen Fällen getäuscht. Abgesehen davon, dass selbst mit GDI die Programmierung wesentlich einfacher als mit GDI+ ist, fehlen schlicht und einfach einige Funktionen, andere sind so langsam, dass man glaubt, der alte 486er aus dem Keller sei wiederauferstanden.

Aus diesem Grund haben wir uns entschlossen, GDI in das vorliegende Buch aufzunehmen, auch wenn Sie damit **keinen managed Code** produzieren.

Da Sie nur schwer GDI+ und GDI-Funktionen "mischen" können (das *Graphics*-Objekt ist während der Arbeit mit den GDI-Funktionen gesperrt), bleibt Ihnen bei der Verwendung von GDI-Funktionen meist nur der komplette Umstieg während einer bestimmten Grafikoperation (z.B. Zeichnen eines Diagramms). Aus diesem Grund stellen wir Ihnen GDI noch einmal im Zusammenhang vor, auch wenn Ihnen einige Funktionen aus dem bisherigen Kapitel bekannt vorkommen dürften.

36.5.1 Am Anfang war das Handle ...

Wenn Sie bereits in der Online-Hilfe bzw. in der MSDN-Library geschmökert haben, werden Sie im Zusammenhang mit GDI-Funktionen dort auch auf den Begriff *Handle* gestoßen sein. Was also ist ein Handle?

Mit Handle wird eine Nummer (in .NET vom Typ *IntPtr*) bezeichnet, mit der ein Fenster, Steuerelement etc. eindeutig gekennzeichnet ist. Jedes Fenster bzw. jede Komponente in C# verfügt über diese Eigenschaft.

Neben dem Handle finden Sie noch die Bezeichnungen *Gerätekontext* (*DC*) bzw. *Handle auf den Gerätekontext* (*HDC*). Dieser Gerätekontext ist die Schnittstelle zwischen Ihrem Anwendungsprogramm und den eigentlichen Ausgabegeräten wie Bildschirm, Fenster, Drucker etc. Der Kontext eines Gerätes beinhaltet nicht nur alle Informationen, Eigenschaften und Attribute, sondern auch die Funktionen, um dieses Gerät anzusprechen.

Daraus folgt auch der prinzipielle Ablauf beim Arbeiten mit den GDI-Funktionen:

- Anlegen eines Gerätekontextes
- Anlegen von Objekten
- Zuweisen der neuen Objekte
- Grafikausgaben (mit obigen Objekten)
- Wiederherstellen der Ursprungsobjekte
- Zerstören der neuen Objekte
- Freigabe bzw. Schließen des Gerätekontextes

HINWEIS: Für die Verwendung aller GDI-Funktionen gilt, dass Sie diese vorher als Klassenmethoden (*DllImport*-Attribut) in Ihr C#-Programm eingebunden haben. Das betrifft ebenfalls alle verwendeten Konstanten.

Doch genug der Theorie, ein praktisches Beispiel wird Ihnen die Augen öffnen.

BEISPIEL 36.38: Grafikausgabe über einen Gerätekontext (Screen)

C#

1. Binden Sie zunächst die nötigen GDI-Funktionen in das Hauptformular ein:

```
using System.Runtime.InteropServices;
...
    [DllImport("user32.dll")]
    private  static extern int  GetDC(int  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(int  hwnd,int  hdc);

    [DllImport("gdi32.dll")]
    private  static extern int  CreatePen(int  nPenStyle,int  nWidth,int  crColor);

    [DllImport("gdi32.dll")]
    private  static extern int  SelectObject(int  hdc,int  hObject);

    [DllImport("gdi32.dll")]
```

BEISPIEL 36.38: Grafikausgabe über einen Gerätekontext (Screen)

```
private  static extern int  SetPixel(int  hdc,int  x,int  y,int  crColor);

[DllImport("gdi32.dll")]
private  static extern int  LineTo(int  hdc,int  x,int  y);
[DllImport("gdi32.dll")]
private  static extern int  DeleteObject(int  hObject);
```

2. Anlegen eines Gerätekontexts, der Rückgabewert ist ein Handle auf den DC:

```
int dc , hp, ho, dx, dy, i, l;

Random rnd = new Random();
dc = GetDC(0);
dx = 1000;
dy = 500;
```

3. Erstellen eines Stifts (Objekt), der Rückgabewert ist ein Handle:

```
hp = CreatePen(0, 10, 0);
```

4. Auswahl des neuen Stifts über das Handle, der Rückgabewert ist das Handle auf den vorherigen, nun ungültigen Stift:

```
ho = SelectObject(dc, hp);
```

5. Diverse Grafikausgaben vornehmen:

```
for (i = 0; i <50; i++)
{
  l = SetPixel(dc, rnd.Next(dx) , rnd.Next(dy), 0);
  l = LineTo(dc, rnd.Next(dx), rnd.Next(dy));
}
```

6. Wiederherstellen der alten Objekte (dazu haben wir uns den Wert gemerkt!) und Löschen der erstellten Objekte:

```
DeleteObject(SelectObject(dc, ho));
```

7. Löschen des Gerätekontextes:

```
ReleaseDC(0,dc);
```

Was mit obigem Beispiel möglich ist, das direkte Zeichnen auf dem Desktop, werden Sie nicht mit GDI+ realisieren können.

HINWEIS: Generell gilt für alle GDI-Aufrufe die Maßeinheit Pixel!

36.5.2 Gerätekontext (Device Context Types)

In C# führen zwei verschiedene Wege zum HDC. Zum einen können Sie einen vom *Graphics*-Objekt bereitgestellten DC verwenden (Methode *GetHDc*), zum anderen können Sie auch einen eigenen DC erzeugen. Welche Variante zu bevorzugen ist, lässt sich nicht eindeutig beantworten.

Einerseits ist die Arbeit mit C#-DCs recht komfortabel und sicher (Sie vergessen nie das Löschen des DC, da Sie erst wieder den DC mit *ReleaseHdc* freigeben müssen!), andererseits kommen Sie in bestimmten Fällen nicht um die Definition eines eigenen DCs herum, z.B. wenn Sie einen DC brauchen, den C# nicht zur Verfügung stellt (Screen), oder wenn Sie mit einigen bestimmten GDI-Funktionen arbeiten.

Windows unterstützt vier verschiedene Typen von Gerätekontexten:

Gerätekontext	Beschreibung
Display	Ausgaben auf dem Bildschirm, d.h. Fenster, Controls oder Screen. Erzeugen mit *GetDC*, *GetDCEx* oder *BeginPaint* Löschen mit *ReleaseDC* oder *EndPaint*
Printer	Drucker/Plotter der verschiedenen Typen. Erzeugen mit *CreateDC* Löschen mit *DeleteDC*
Memory	Speicherabbilder von Bitmaps. Diese sind nicht sichtbar, können aber das Ziel von Grafikausgaben sein. Erzeugen mit *CreateCompatibleDC* Löschen mit *DeleteDC*
Information	... dient lediglich der Informationsgewinnung (z.B. im Zusammenhang mit dem Drucker), es können keine Grafikausgaben vorgenommen werden (*GetDeviceCaps*). Erzeugen mit *CreateIC* Löschen mit *DeleteDC*

Formular-DC über Graphics-Objekt erzeugen

Wie Sie beispielsweise den DC des Screens mit *GetDC* ermitteln, wurde bereits im vorhergehenden Abschnitt gezeigt. Wie Sie mit *GetHDc* den DC des Formulars abfragen, zeigt das folgende Beispiel.

BEISPIEL 36.39: Verwendung von *Graphics.GetHDc*

C#

```
using System.Runtime.InteropServices;
...
   [DllImport("gdi32.dll")]
   private  static extern int LineTo(IntPtr  hdc, int  x,int  y);
```

Achten Sie auf die obige DC-Deklaration als *IntPtr*, nur so können Sie problemlos den C#-DC verwenden.

...

BEISPIEL 36.39: Verwendung von *Graphics.GetHDc*

```
private void button1_Click(object sender, EventArgs e)
{
  Graphics g;
  IntPtr mydc;
  g = this.CreateGraphics();     // Graphics-Objekt erzeugen
  mydc = g.GetHdc();             // DC abfragen
  LineTo(mydc, 0, 0);            // GDI-Funktionen verwenden ...
  LineTo(mydc, 100, 100);
  g.ReleaseHdc(mydc);            // DC freigeben
}
```

Sie können natürlich auch gleich das im *Paint*-Ereignis bereitgestellte Objekt *e.Graphics* verwenden.

Formular-DC mit GetDC erzeugen

Über die Funktion *GetDC*, die ein Fenster-Handle als Parameter erwartet, können Sie ebenfalls einen DC erzeugen.

BEISPIEL 36.40: Verwendung von *GetDC*

C#
```
using System.Runtime.InteropServices;
...
    [DllImport("gdi32.dll")]
    private  static extern int LineTo(IntPtr  hdc, int  x, int  y);

    [DllImport("user32.dll")]
    private  static extern IntPtr GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int ReleaseDC(IntPtr  hwnd, IntPtr  hdc);
...
    private void button1_Click(object sender, EventArgs e)
    {
      IntPtr mydc;
      mydc = GetDC(this.Handle);
      LineTo(mydc, 0, 0);           // GDI-Funktionen verwenden ...
      LineTo(mydc, 100, 100);
      ReleaseDC(this.Handle, mydc);
    }
```

HINWEIS: Die gleiche Vorgehensweise ist bei allen Steuerelementen möglich, die ein Fenster-Handle bereitstellen.

36.5.3 Koordinatensysteme und Abbildungsmodi

Bisher sind Sie wahrscheinlich nur mit dem C#-eigenen Koordinatensystem in Berührung gekommen, dessen Nullpunkt sich zunächst in der linken oberen Ecke befindet. Ähnlich wie bei GDI+ können auch bei GDI verschiedene Skalierungsmodi sowie Koordinatensysteme gewählt werden. Zum Einsatz kommt die Funktion *SetMapMode*, mit der Sie einem Ausgabegerät (bzw. dessen DC) ein neues Koordinatensystem bzw. einen anderen Abbildungsmodus zuweisen können.

Für die Druckausgabe ist insbesondere der Abbildungsmodus *MM_LOMETRIC* interessant, in dem Sie unabhängig von der Druckerauflösung mit Millimetern arbeiten können.

BEISPIEL 36.41: Drucken eines 10 cm großen Rechtecks

C#
```
using System.Runtime.InteropServices;
...
    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);

    private const int MM_LOMETRIC = 2;

    [DllImport("gdi32.dll")]
    private  static extern int  Rectangle(IntPtr  hdc,int  X1,int  Y1,int  X2,int  Y2);
...
    private void printDocument1_PrintPage(object sender, Printing.PrintPageEventArgs e)
    {
      IntPtr mydc;
      mydc = e.Graphics.GetHdc();
      SetMapMode(mydc, MM_LOMETRIC);
      Rectangle(mydc, 100, -100, 1100, -1100);
      e.Graphics.ReleaseHdc(mydc);
    }
```

HINWEIS: Beachten Sie, dass die y-Koordinaten negativ angegeben werden müssen!

Die folgende Tabelle listet die zulässigen Konstanten auf.

Typ	Beschreibung
MM_ANISOTROPIC	Virtuelle Einheiten werden in willkürliche Einheiten mit willkürlich skalierten Achsen umgewandelt. Das Setzen des Abbildungsmodus verändert die aktuellen Fenster- und Grafikfenster-Einstellungen nicht. Zum Ändern der Einheiten für Orientierung und Skalierung sollte Ihre Anwendung die Funktionen *SetWindowExtEx* und *SetViewportExtEx* verwenden.
MM_HIENGLISH	Jede virtuelle Einheit wird in 0.001 Zoll umgewandelt. Positive x sind rechts, positive y oben. Beachten Sie, dass es bei diesem Abbildungsmodus sehr schnell zu Wertebereichsüberschreitungen kommen kann.

Typ	Beschreibung
MM_HIMETRIC	Eine virtuelle Einheit wird in 0.01 Millimeter umgewandelt. Positive x sind rechts, positive y oben.
MM_ISOTROPIC	Virtuelle Einheiten werden in willkürliche Einheiten mit gleich skalierten Achsen umgewandelt, d. h., eine Einheit auf der x-Achse entspricht einer Einheit auf der y-Achse. Um die gewünschten Einheiten und Achsenorientierungen zu bestimmen, müssen Sie die Funktionen *SetWindowExtEx* und *SetViewportExtEx* verwenden.
MM_LOENGLISH	Jede virtuelle Einheit wird in 0.01 Zoll umgewandelt. Positive x sind rechts, positive y oben.
MM_LOMETRIC	Eine virtuelle Einheit wird in 0.1 Millimeter umgewandelt. Positive x sind rechts, positive y oben.
MM_TEXT	Eine virtuelle Einheit wird in einen (1) Geräte-Bildpunkt gewandelt. Positive x sind rechts, positive y unten.
MM_TWIPS	Eine virtuelle Einheit wird in 1/20 Punkt umgewandelt. Da 1 Punkt gleich 1/72 Zoll ist, entspricht ein Twip 1/1440 Zoll. Positive x sind rechts, positive y oben.

Für die Ausgabe auf dem Bildschirm sind die Modi *MM_ANISOTROPIC* und *MM_ISOTROPIC* interessant, können Sie doch damit beliebige Koordinatensysteme realisieren. Allerdings ist der Modus *MM_ISOTROPIC* dahingehend eingeschränkt, dass x- und y-Achse im gleichen Maßstab skaliert werden, die Abbildung also proportional vergrößert bzw. verkleinert wird.

Die Funktionen *SetWindowExtEx* und *SetViewportExtEx* brauchen Sie für die Auswahl der Skalierungsfaktoren.

BEISPIEL 36.42: Eine technische Zeichnung mit den virtuellen Abmessungen 4000 x 4000 soll in den Clientbereich eines Formulars skaliert werden.

C#
```
using System.Runtime.InteropServices;
...
    private const int MM_ISOTROPIC = 7;

    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);

    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowExtEx(IntPtr  hdc,int  nX,int  nY, int lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  SetViewportExtEx(IntPtr  hdc,int  nX,int  nY, int lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  Rectangle(IntPtr  hdc,int  X1,int  Y1,int  X2,int  Y2);

    [DllImport("user32.dll")]
```

BEISPIEL 36.42: Eine technische Zeichnung mit den virtuellen Abmessungen 4000 x 4000 soll in den Clientbereich eines Formulars skaliert werden.

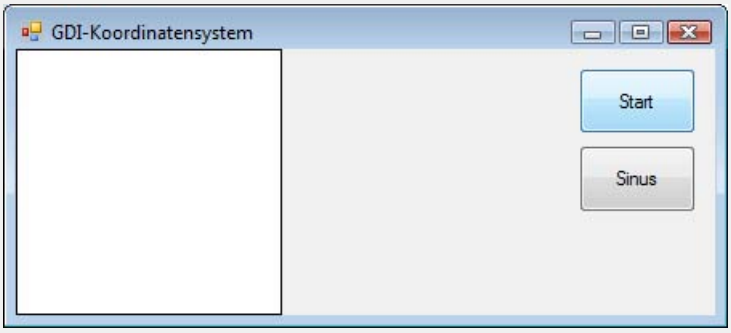
```
private  static extern IntPtr  GetDC(IntPtr  hwnd);

[DllImport("user32.dll")]
private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);

private void button1_Click(object sender, EventArgs e) {
  IntPtr mydc;
  mydc = GetDC(this.Handle);
  SetMapMode(mydc, MM_ISOTROPIC);
  SetWindowExtEx(mydc, 4000, 4000, 0);
  SetViewportExtEx(mydc, this.ClientSize.Width, this.ClientSize.Height, 0);
  Rectangle(mydc, 0, 0, 4000, 4000);    // ... zeigt die Größe der Zeichnung
  ReleaseDC(this.Handle, mydc);
}
```

Nach der Auswahl des Abbildungsmodus (natürlich verwenden wir *MM_ISOTROPIC*, die Zeichnung soll ja nicht verzerrt werden), übergeben wir der Funktion *SetWindowExtEx* die virtuellen Maximalabmessungen unseres neuen Fensters. Die Größe des Viewports entspricht den realen Innenabmessungen des Formulars (*ScaleWidth*, *ScaleHeight*).

Alle weiteren Grafikausgaben (Text, Linien etc.) werden jetzt so skaliert, als ob das Fenster eine Innenabmessung von 4000 x 4000 Pixel hätte.

Ergebnis

Wie Sie der folgenden Abbildung entnehmen können, wird das Rechteck dank *MM_ISOTROPIC* unverzerrt dargestellt:



Möchten Sie positive y-Werte nach oben abtragen, können Sie auch folgende Zuweisung für *SetViewportExtEx* verwenden:

```
SetViewportExtEx(mydc, this.ClientSize.Width, -this.ClientSize.Height, 0);
```

Mit Hilfe der Funktion *SetWindowOrgEx* ist es weiterhin möglich, den Nullpunkt des Koordinatensystems beliebig zu verschieben, Sie brauchen bei späteren Grafikausgaben also nicht extra einen Offset zu addieren.

BEISPIEL 36.43: Zeichnen einer Sinuskurve im Bereich 0 ... 2π.

C#

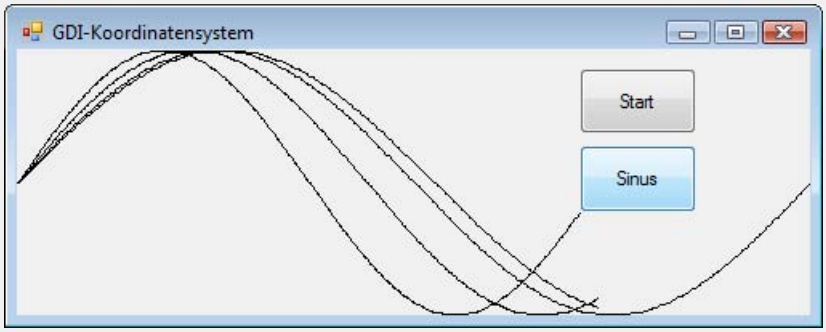
Die Funktion soll so skaliert werden, dass der Clientbereich unabhängig von der Formulargröße voll ausgenutzt wird.

```
using System.Runtime.InteropServices;
...
    [StructLayout(LayoutKind.Sequential)]
      private  struct POINTAPI {
     public  int  x;
     public  int  y;  }

    private const int MM_ANISOTROPIC = 8;
    [DllImport("gdi32.dll")]
    private  static extern int  SetMapMode(IntPtr  hdc,int  nMapMode);

    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowExtEx(IntPtr  hdc,int  nX,int  nY, int lpSize);

    [DllImport("gdi32.dll")]
    private  static extern int  SetViewportExtEx(IntPtr  hdc,int  nX,int  nY, int lpSize);
    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);
    [DllImport("gdi32.dll")]
    private  static extern int  LineTo(IntPtr  hdc,int  x,int  y);

    [DllImport("gdi32.dll")]
    private  static extern int  SetWindowOrgEx(IntPtr  hdc,int  nX,int  nY,
             [MarshalAs(UnmanagedType.Struct)] ref  POINTAPI lpPoint);

    [DllImport("gdi32.dll")]
    private  static extern int  MoveToEx(IntPtr  hdc,int  x,int  y,
             [MarshalAs(UnmanagedType.Struct)] ref  POINTAPI lpPoint);
...
private void button1_Click(object sender, EventArgs e)
 {
  IntPtr mydc;
  POINTAPI p;
  int i;
  Single max;
  p.x = 0; p.y = 0;
  mydc = GetDC(this.Handle);
  max = (int) (2 * 3.1415926 * 100);
  SetMapMode(mydc, MM_ANISOTROPIC);
  SetWindowExtEx(mydc, (int) max, 200, 0);
  SetViewportExtEx(mydc, this.ClientSize.Width, - this.ClientSize.Height, 0);
```

BEISPIEL 36.43: Zeichnen einer Sinuskurve im Bereich 0 ... 2π.

```
  SetWindowOrgEx(mydc, 0, 100, ref p);   // Verschiebung Nullpunkt
  MoveToEx(mydc, 0, 0, ref p);
  for (i = 0; i <= max; i++)
    LineTo(mydc, i, (int) (100 * Math.Sin( i / 100f )));
  ReleaseDC(this.Handle, mydc);
}
```

Ergebnis



Mit den oben genannten Funktionen ist lediglich eine Verschiebung bzw. Skalierung möglich. Wem das nicht reicht, der findet eine weitere GDI-Funktion, die alle mathematisch relevanten Transformationen vornehmen kann. Die Rede ist von *SetWorldTransform*, mit der Sie Verschiebungen, Skalierungen, Rotationen, Spiegelungen und Scherungen realisieren können.

HINWEIS: Das Gleiche gibt es auch in GDI+, siehe Abschnitt 36.1!

36.5.4 Zeichenwerkzeuge/Objekte

Für die Ausgabe von Grafiken (dazu zählt auch Text!) unterstützt Windows-GDI verschiedene grafische Objekte:

- Stifte (Pen)
- Pinsel (Brush)
- Bitmaps
- Paletten
- Schriftarten (Font)
- Regionen (Region)
- Paths

In den folgenden Abschnitten wollen wir kurz auf die wichtigsten Möglichkeiten und Funktionen eingehen.

Erzeugen und Parametrieren der Objekte

Bis auf wenige Ausnahmen müssen Sie alle oben genannten Objekte erst erzeugen (*CreatePen, CreateBrush etc.*), bevor Sie diese nutzen können. Neben dem Erstellen des neuen Objekts müssen Sie sich auch noch um das Sichern der bisherigen Objekte kümmern.

HINWEIS: Die neuen Objekte müssen wieder gelöscht werden, sonst werden wertvolle Systemressourcen verschwendet!

BEISPIEL 36.44: Erzeugen eines neuen Schrifttyps (Font-Objekt)

C#

```
using System.Runtime.InteropServices;
...
    [DllImport("user32.dll")]
    private  static extern IntPtr  GetDC(IntPtr  hwnd);

    [DllImport("user32.dll")]
    private  static extern int  ReleaseDC(IntPtr  hwnd,IntPtr  hdc);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  CreateFont(int  H,int  W,int  E,int  O,int  w,int  I,
             int  u,int  S,int  C,int  OP,int  CP,int  Q,int  PAF,string  F);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  SelectObject(IntPtr  hdc, IntPtr  hObject);

    [DllImport("gdi32.dll")]
    private  static extern IntPtr  DeleteObject(IntPtr  hObject);

    [DllImport("gdi32.dll")]
    private  static extern int  TextOut(IntPtr  hdc,int  x,int  y,string lpstring,
                                        int  nCount);
...
    private void button1_Click(object sender, EventArgs e)
    {
      IntPtr mydc, hFont, fontOld;
      mydc = (IntPtr) GetDC(this.Handle);
```

Neues Objekt erzeugen:

```
  hFont = CreateFont(-40, 0, 45 * 10, 0, 400, 0, 0, 0, 1, 4, 0x10, 2, 4, "Arial");
```

Altes Objekt sichern und neues Objekt setzen:

```
  fontOld = SelectObject(mydc, hFont);
```

GDI-Grafikausgaben:

```
  TextOut(mydc, 100, 100, "Hallo", 5);
```

BEISPIEL 36.44: Erzeugen eines neuen Schrifttyps (Font-Objekt)

Altes Objekt wiederherstellen:

```
SelectObject(mydc, fontOld);
```

Neues Objekt löschen:

```
DeleteObject(hFont);
ReleaseDC(this.Handle, mydc);
}
```

Das Zurücksetzen und Löschen des Objekts können Sie auch mit einer Anweisung realisieren

```
DeleteObject(SelectObject(mydc, fontOld));
```

... da der Rückgabewert der Funktion *SelectObject* der bisher gültige Handle ist.

HINWEIS: Eine weitere Variante, Objekte zu erzeugen, bietet sich mit der Funktion *GetStockObject*, über die bereits vordefinierte Systemobjekte abgerufen werden können. Diese Objekte brauchen nicht gelöscht zu werden.

36.5.5 Bitmaps

Eine Bitmap besteht neben dem Speicherabbild (Array aller Pixel) aus einem Header, der Größe und Auflösung beschreibt, sowie einer Palette. Windows unterstützt zwei Arten von Bitmaps:

- geräteabhängige (DDBs[1])
- geräteunabhängige (DIBs[2])

Bei geräteabhängigen Bitmaps wird die Grafik so im Arbeitsspeicher abgelegt, wie es das Ausgabegerät erfordert. Im Unterschied dazu werden geräteunabhängige Bitmaps in einem fest definierten Format abgelegt, also unabhängig vom Ausgabegerät.

An dieser Stelle wollen wir nicht weiter auf die Arbeit mit den Systemfunktionen eingehen, bietet doch C# im Zusammenhang mit Bitmaps fast schon alles, was der Programmierer so braucht. Bis auf eine Ausnahme: Wollen Sie Screenshots von einzelnen Fenstern bzw. vom gesamten Screen erzeugen, müssen Sie sich zwangsläufig mit einigen GDI-Funktionen beschäftigen.

BEISPIEL 36.45: Einen Screenshot realisieren

C#
```
using System.Runtime.InteropServices;
...
private int SRCCOPY = 0xCC0020;

    [DllImport("gdi32.dll")]
    private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,
```

[1] Device-Dependent Bitmap
[2] Device-Independent Bitmap

BEISPIEL 36.45: Einen Screenshot realisieren

```
            int  nHeight,IntPtr  hSrcDC,int  xSrc,int  ySrc,int  dwRop);

   [DllImport("user32.dll")]
   private  static extern IntPtr  GetDC(int  hwnd);
   [DllImport("user32.dll")]
   private  static extern int  ReleaseDC(int  hwnd,IntPtr  hdc);
...
     Graphics g1;
     IntPtr dc1, dc2;
     Image img;
```

Zunächst ein passendes Bitmap-Objekt in C# erstellen (Screengröße):

```
  img = new Bitmap(Screen.PrimaryScreen.WorkingArea.Width,
                   Screen.PrimaryScreen.WorkingArea.Height);
```

Ein *Graphics*-Objekt zuordnen:

```
g1 = Graphics.FromImage(img);
```

Den DC des Desktops ermitteln:

```
dc1 = GetDC(0);
```

Den DC unserer (leeren) Bitmap ermitteln:

```
dc2 = g1.GetHdc();
```

Mittels *BitBlt* die Bitmap-Inhalte des Desktops in unserer C#-Bitmap kopieren:

```
BitBlt(dc2, 0, 0, Screen.PrimaryScreen.WorkingArea.Width,
       Screen.PrimaryScreen.WorkingArea.Height, dc1, 0, 0, SRCCOPY);
```

Desktop-DC freigeben:

```
ReleaseDC(0, dc1);
```

Bitmap-DC freigeben:

```
g1.ReleaseHdc(dc1);
```

Im Weiteren können Sie jetzt alle C#-Möglichkeiten nutzen, beispielsweise kann die Bitmap im PNG-Format gespeichert werden:

```
img.Save("c:\\Screen.png", Drawing.Imaging.ImageFormat.Png);
...
```

Einzelheiten zu den verschiedenen DCs sind in folgender Tabelle zusammengefasst:

Gerätekontext	Quellcode
DC gesamtes Fenster	*dc* = GetWindowDC(hWnd); *ReleaseDC(0, dc);*

Gerätekontext	Quellcode
DC Clientbereich des Fensters	*dc* = GetDC(hWnd); *ReleaseDC(hWnd, dc);*
DC Bildschirm	*dc = GetDC(0);* *ReleaseDC(0, dc);*

Bitmaps kopieren

Zum Kopieren von Bitmaps verwenden Sie die *BitBlt*-Funktion. Ziel und Quelle können identisch sein.

```
SYNTAX: [DllImport("gdi32.dll")]
        private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,int
                                    nHeight,IntPtr  hSrcDC,int  xSrc,int  ySrc,int  dwRop);
```

Der wichtigste Parameter ist *dwRop*, mit dem Sie die Art der Verknüpfung von Quelle und Ziel festlegen:

Parameter	Beschreibung
BLACKNESS	... schaltet alle Ausgaben auf Schwarz.
DSTINVERT	... invertiert die Ziel-Bitmap.
MERGECOPY	... verknüpft das Muster mit der Quell-Bitmap unter Verwendung der AND-Operation.
MERGEPAINT	... verknüpft die invertierte Quell-Bitmap mit der Ziel-Bitmap unter Verwendung der OR-Operation.
NOTSRCCOPY	... kopiert die invertierte Quell-Bitmap in die Ziel-Bitmap.
NOTSRCERASE	... invertiert das Ergebnis der OR-Verknüpfung von Quell- und Ziel-Bitmap.
PATCOPY	... kopiert das Muster in die Ziel-Bitmap.
PATINVERT	... verknüpft die Ziel-Bitmap mit dem Muster unter Verwendung der XOR-Operation.
PATPAINT	... verknüpft die invertierte Quell-Bitmap mit dem Muster unter Verwendung der OR-Bedingung, verknüpft das Ergebnis dieses Vorgangs mit der Ziel-Bitmap unter Verwendung der OR-Operation.
SRCAND	... verknüpft Pixel der Quell- und Ziel-Bitmaps unter Verwendung der AND-Operation.
SRCCOPY	... kopiert die Quell-Bitmap auf die Ziel-Bitmap.
SRCERASE	... invertiert die Ziel-Bitmap und verknüpft das Ergebnis mit der Quell-Bitmap unter Verwendung der AND-Operation.
SRCINVERT	... verknüpft Pixel von Quell- und Ziel-Bitmaps unter Verwendung der XOR-Operation.

Parameter	Beschreibung
SRCPAINT	... verknüpft Pixel von Quell- und Ziel-Bitmaps unter Verwendung der OR-Operation.
WHITENESS	... schaltet alle Ausgaben auf Weiß.

HINWEIS: Die Werte für oben genannte Parameter können Sie auch im Zusammenhang mit der *StretchBlt*-Funktion verwenden.

Natürlich sind die meisten der Optionen nur mit zwei verschiedenen Bitmaps sinnvoll anwendbar.

Bitmaps skalieren

Obwohl das Skalieren von Bitmaps nicht allzu sinnvoll ist (Ausgabequalität!), finden sich doch einige Anwendungsgebiete. Möchten Sie beispielsweise eine Bitmap in bestimmter Größe auf dem Drucker ausgeben (10 x 10 cm), bleibt Ihnen meist nichts anderes übrig als die Skalierung der Bitmap. Das GDI bietet dazu die Funktion *StretchBlt*:

SYNTAX:
```
[DllImport("gdi32.dll")]
private  static extern int  StretchBlt(IntPtr hDestDC,int x,int y,int  nWidth,int
                            nHeight, IntPtr hSrcDC,int xSrc,int  ySrc,int  nSrcWidth,
                            int  nSrcHeight,int  dwRop);
```

BEISPIEL 36.46: Durch geschickte Wahl der Koordinaten lässt sich eine Bitmap in horizontaler und vertikaler Richtung spiegeln.

C#

Spiegeln vertikal:

```
StretchBlt(myDC1, 0, 0, Breite, Höhe, myDC2, 0, Höhe, Breite, Höhe, SRCCOPY);
```

Spiegeln horizontal:

```
Stretchblt(myDC1, 0, 0, Breite, Höhe, myDC2, Breite, 0, -Breite, Höhe, SRCCOPY);
```

HINWEIS: Sind Quell- und Ziel-Bitmap gleich groß, sollten Sie zum Kopieren besser die *BitBlt*-Funktion verwenden, da diese schneller ist.

Damit wollen wir die Ausführungen zu GDI beenden, auch wenn noch weitere interessante Themen wie XOR-Zeichenmodus etc. damit möglich sind. Die grundsätzliche Vorgehensweise dürfte dennoch klar geworden sein.

36.6 Praxisbeispiele

36.6.1 Die Transformationsmatrix verstehen

Wem die Ausführungen in Abschnitt 36.1 zu theoretisch waren, hier ist ein praktisches Beispiel zum Experimentieren.

Über sechs *TextBox*en erhalten Sie die Möglichkeit, die Matrix-Elemente *m11* bis *dy* zur Laufzeit frei zu verändern. Einige weitere, bereits vorkonfigurierte Beispiele können Sie über eine *ComboBox* abrufen.

Oberfläche

Siehe Laufzeitansicht am Schluss des Beispiels. In die *ComboBox* tragen Sie folgende Werte ein:

```
-
Reset
Rotate(15)
Rotate(45)
Shear(0.1,0)
Shear(0,0.1)
Translate(5,0)
Translate(0,5)
Scale(2,0)
Scale(0,2)
```

Quelltext

Binden Sie zunächst den Namespace *System.Drawing.Drawing2D* ein.

Das Button-Klick-Ereignis:

```
Matrix m = new Matrix(Convert.ToSingle(textBox1.Text), Convert.ToSingle(textBox2.Text),
                      Convert.ToSingle(textBox3.Text), Convert.ToSingle(textBox4.Text),
                      Convert.ToSingle(textBox5.Text), Convert.ToSingle(textBox6.Text));
```

Wie Sie sehen, füllen wir die Matrix gleich beim Erstellen. Auf eine Fehlerprüfung haben wir verzichtet.

```
Graphics g = this.CreateGraphics();
```

Ausgabe löschen und zunächst ein Rechteck **ohne** Transformation zeichnen:

```
g.Clear(this.BackColor);
g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Transformations-Matrix anwenden:

```
g.Transform = m;
```

Die eigentlichen Zeichenfunktionen realisieren:

```
g.DrawLine(Pens.Red, 0, 0, 200, 0);
```

```
g.DrawLine(Pens.Red, 0, 0, 0, 200);
g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
```

Die über die *ComboBox* abrufbaren Beispiele:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
    Matrix m;
```

Erster Eintrag (ohne Funktion):

```
    if (comboBox1.SelectedIndex == 0) return;
```

Zweiter Eintrag (Zurücksetzen der Matrix):

```
    if (comboBox1.SelectedIndex == 1)
    {
        m = new Matrix(1, 0, 0, 1, 0, 0);
    }
```

Auslesen der *TextBox*en:

```
    else
    {
        m = new Matrix(Convert.ToSingle(textBox1.Text),
                       Convert.ToSingle(textBox2.Text),
                       Convert.ToSingle(textBox3.Text),
                       Convert.ToSingle(textBox4.Text),
                       Convert.ToSingle(textBox5.Text),
                       Convert.ToSingle(textBox6.Text));
    }
    Graphics g = this.CreateGraphics();
    g.Clear(this.BackColor);
    g.DrawRectangle(Pens.Red, 10, 10, 50, 100);
```

Direktes Verändern der Matrix durch Methodenaufrufe:

```
    switch (comboBox1.SelectedIndex)
    {
      case 2:   // Rot 15
          m.Rotate(15);
          break;
      case 3:   // Rot 45
          m.Rotate(45);
          break;
      case 4:
          m.Shear(0.1f, 0);
          break;
      case 5:
          m.Shear(0, 0.1f);
          break;
```

```
            case 6:
                m.Translate(5, 0);
                break;
            case 7:
                m.Translate(0, 5);
                break;
            case 8:
                m.Scale(2, 0);
                break;
            case 9:
                m.Scale(0, 2);
                break;
            default:
                  break;
        }
```

Die neuen Werte in die *TextBox*en eintragen:

```
        textBox1.Text = m.Elements[0].ToString();
        textBox2.Text = m.Elements[1].ToString();
        textBox3.Text = m.Elements[2].ToString();
        textBox4.Text = m.Elements[3].ToString();
        textBox5.Text = m.Elements[4].ToString();
        textBox6.Text = m.Elements[5].ToString();
```

Transformationsmatrix zuordnen:

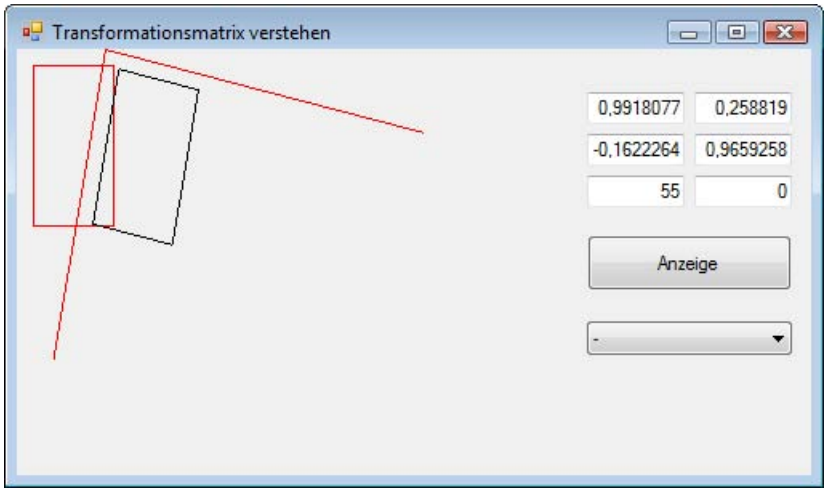
```
        g.Transform = m;
```

Grafikausgabe:

```
        g.DrawLine(Pens.Red, 0, 0, 200, 0);
        g.DrawLine(Pens.Red, 0, 0, 0, 200);
        g.DrawRectangle(Pens.Black, 10, 10, 50, 100);
        comboBox1.SelectedIndex = 0;
    }
```

Test

Starten Sie die Anwendung und versuchen Sie sich zunächst an Skalierungen und Translationen (siehe Abschnitt 36.1). Wer zunächst einen Überblick bekommen möchte, setzt die Matrix über die *ComboBox* zurück und wählt dann eine Transformation per *ComboBox*. Die Änderung der Werte in den *TextBox*en dürften schnell für mehr Klarheit sorgen:



36.6.2 Eine 3D-Grafikausgabe in Aktion

Nachdem wir Sie im Abschnitt 36.4 mit endlosen Ausführungen zur 3D-Grafik traktiert haben, möchten Sie die Algorithmen sicher auch in Aktion erleben.

Oberfläche

Fügen Sie entsprechend der Abbildung am Ende des Beispiels Schaltflächen für die Skalierung, die Translation und die Rotation ein:

Der *TrackBar* bestimmt die Refresh-Geschwindigkeit (ein *Timer* ist auch noch einzufügen).

Quelltext

Auf die Algorithmen und die Klasse *c3D* sind wir ja bereits im Abschnitt 36.4 des Kapitels eingegangen. An dieser Stelle beschränken wir uns auf das Anzeigeprogramm.

```
public partial class Form1 : Form
{
```

Eine Instanz unserer 3D-Klasse bilden:

```
    c3D my3DSystem = new c3D();
```

In dieser Variablen speichern wir die angeklickte Schaltfläche (wird im *Timer* ausgewertet):

```
    object cmd;
```

Im Konstruktor kümmern wir uns zunächst um die Anzeigequalität (Flimmern verringern):

```
    public Form1()
    {
        InitializeComponent();
        SetStyle(ControlStyles.UserPaint, true);
        SetStyle(ControlStyles.AllPaintingInWmPaint, true);
```

```
            SetStyle(ControlStyles.DoubleBuffer, true);
        }
```

Die eigentliche Anzeigeroutine fällt recht kurz aus:

```
        protected override void OnPaint(PaintEventArgs e)
        {
            my3DSystem.Draw(e.Graphics);
        }
```

Wird auf eine der Tasten geklickt, starten wir den *Timer*:

```
        private void button1_MouseDown(object sender, MouseEventArgs e)
        {
            cmd = sender;
            timer1.Start();
        }
```

Beim Loslassen halten wir den *Timer* an:

```
        private void button1_MouseUp(object sender, MouseEventArgs e)
        {
            timer1.Stop();
        }
```

Die Refresh-Geschwindigkeit verändern:

```
        private void trackBar1_Scroll(object sender, EventArgs e)
        {
            timer1.Interval = trackBar1.Value;
        }
```

Ach ja, die zu manipulierenden Objekte müssen wir ja auch noch definieren:

```
        private void Form1_Load(object sender, EventArgs e)
        {
            my3DSystem.AddObject(new point3D(0, 0, 0));
            my3DSystem.AddObject(new coordinateSystem3D(3));
            my3DSystem.AddObject(new line3D(-2, 1, 2, 1, 1, 2));
            my3DSystem.AddObject(new line3D( -2, 1, 2, -2, -1, 2));
...
            my3DSystem.AddObject(new line3D(0.5f, 0.5f, -1, 0.5f, -1, -1));
            my3DSystem.Scale(50);
        }
```

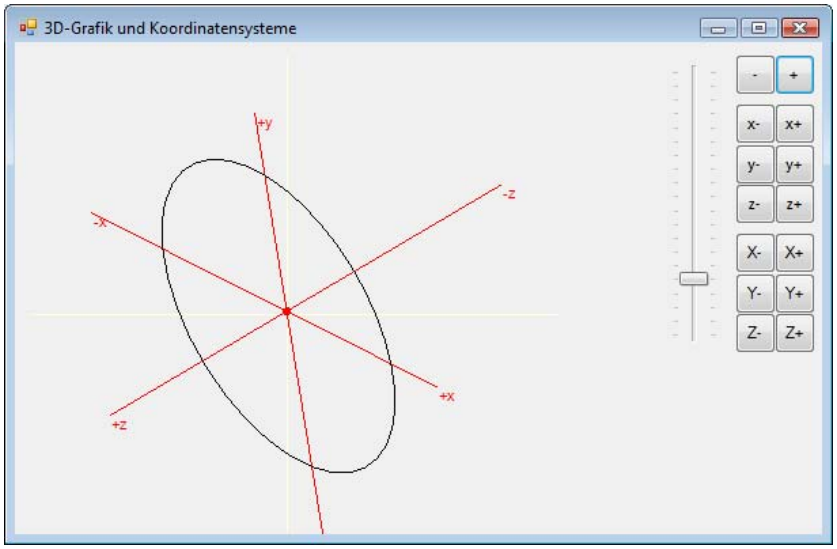
Der *Timer* kümmert sich um die Transformationen:

```
        private void timer1_Tick(object sender, EventArgs e)
        {
            switch ( (cmd as Button).Text)
            {
                case "+":
                    my3DSystem.Scale(1.1f);
                    break;
```

```
...
                case "Z+":
                    my3DSystem.Rotate(0, 0, 1);
                    break;
                default:
                    break;
            }
            this.Invalidate();
        }
```

Test/Bemerkungen

Natürlich können Sie das Programm zunächst so testen wie es ist, besser ist es jedoch, wenn Sie sich auch mal daran versuchen, ein anderes Objekt in unserem 3D-Testlabor anzuzeigen.



Die Lösung:

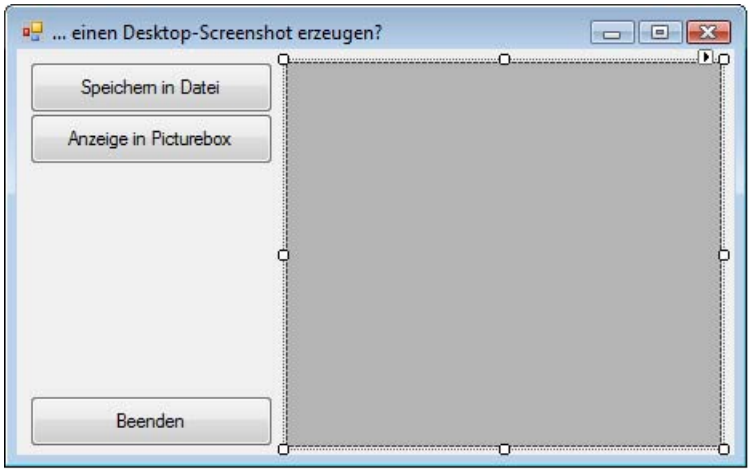
```
            Single xold, yold, xnew, ynew, r;

            r = 2;
            xold = r;
            yold = 0;
            for (int i = 0; i <= 360; i += 10)
            {
                xnew = Convert.ToSingle(r * Math.Cos(i * Math.PI / 180));
                ynew = Convert.ToSingle(r * Math.Sin(i * Math.PI / 180));
                my3DSystem.AddObject(new line3D(xold, yold, 0, xnew, ynew, 0));
                xold = xnew;
                yold = ynew;
            }
```

36.6.3 Einen Fenster-Screenshot erzeugen

Geht es darum, einen Screenshot vom aktuellen Fenster zu erzeugen, kommen Sie um ein wenig GDI-Programmierung nicht herum.

Oberfläche

Erstellen Sie zunächst eine Oberfläche entsprechend folgender Abbildung (*PictureBox*, drei *Buttons*):



Quelltext

```
using System.Runtime.InteropServices;

public partial class Form1 : Form
{
    ...
```

Binden Sie nachfolgende Konstante sowie die GDI-Funktion *BitBlt* ein:

```
    private const int SRCCOPY = 0xCC0020;

    [DllImport("gdi32.dll")]
    private  static extern int  BitBlt(IntPtr  hDestDC,int  x,int  y,int  nWidth,
                                       int  nHeight, IntPtr  hSrcDC,int  xSrc,
                                       int  ySrc,int  dwRop);
    ...
```

Die Routine zum Speichern des Screenshots:

```
    private void Button1_Click(object sender, EventArgs e)
    {
        Graphics g1, g2;
        IntPtr dc1, dc2;
```

```
        Image img;
```

Erzeugen einer neuen Bitmap mit den Maßen und der Farbtiefe des aktuellen Fensters:

```
        g1 = this.CreateGraphics();
        img = new Bitmap(this.ClientRectangle.Width, this.ClientRectangle.Height, g1);
        g2 = Graphics.FromImage(img);
```

Kopieren der Fenster-Bitmap in die eigene Bitmap:

```
        dc1 = g1.GetHdc();
        dc2 = g2.GetHdc();
        BitBlt(dc2, 0, 0, this.ClientRectangle.Width,
               this.ClientRectangle.Height, dc1, 0, 0, 13369376);
        g1.ReleaseHdc(dc1);
        g2.ReleaseHdc(dc2);
```

Speichern der Daten im PNG-Format:

```
        img.Save("c:\\Form1.png", System.Drawing.Imaging.ImageFormat.Png);
        MessageBox.Show("Fenster-Screenshot gesichert", "Info");
    }
```

Ähnlich gestaltet sich die Routine zur Anzeige in der *PictureBox*:

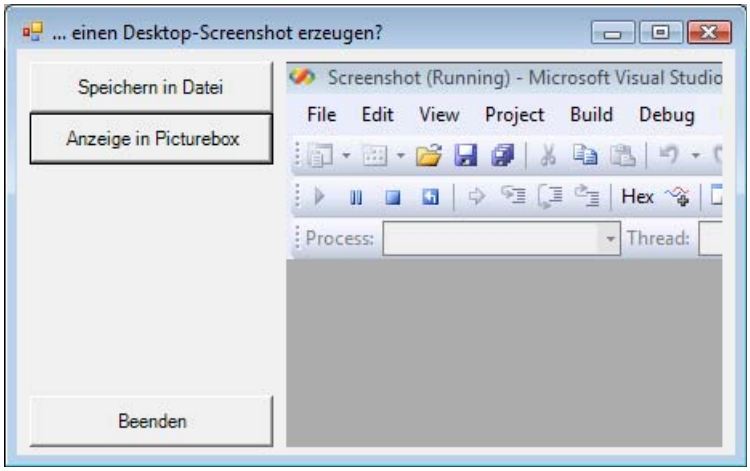
```
    private void Button2_Click(object sender, EventArgs e)
    {
       Graphics g1, g2;
       IntPtr dc1, dc2;
       Image img;

       g1 = this.CreateGraphics();
       img = new Bitmap(this.ClientRectangle.Width, this.ClientRectangle.Height, g1);
       g2 = Graphics.FromImage(img);

       dc1 = g1.GetHdc();
       dc2 = g2.GetHdc();
       BitBlt(dc2, 0, 0, this.ClientRectangle.Width, this.ClientRectangle.Height,
              dc1, 0, 0, 13369376);
       g1.ReleaseHdc(dc1);
       g2.ReleaseHdc(dc2);
```

Wir weisen das *Image* der *PictureBox* zu:

```
        pictureBox1.Image = img;
    }
    ...
}
```

Test

Nach dem Programmstart und dem Klick auf den Button "Anzeige in Picturebox" sollte sich Ihnen der folgende Anblick bieten:



Die erzeugte Datei *Form1.png* findet sich im Verzeichnis "C:\", Sie können diese zum Beispiel mit dem Internet Explorer anzeigen oder aber auch mit der in Windows integrierten Bild- und Fax-anzeige.

Bemerkung

Etwas aufwändiger als ein Fenster-Screenshot ist ein echter Screenshot des gesamten Desktops. Eine entsprechende Lösung finden Sie in unserem [Visual C# 2012 Kochbuch].

Kapitel 37

Ressourcen/Lokalisierung

Über Ressourcen können Sie externe Informationen in Ihr Programm aufnehmen. Das betrifft Texte, Grafiken und andere Elemente, die sich nicht ohne weiteres per Code darstellen lassen. Das .NET-Framework kennt prinzipiell zwei Typen von Ressourcen:

- Manifestressourcen und
- typisierte Ressourcen.

Mit beiden Ressourcentypen und ihren Ableitungen (streng typisierte Ressourcen) werden wir uns in diesem Kapitel auseinandersetzen.

Eine besondere Bedeutung haben typisierte Ressourcen vor allem im Zusammenhang mit der Lokalisierung von Anwendungen (siehe Abschnitt 37.4).

37.1 Manifestressourcen

Manifestressourcen können nahezu beliebige Dateien sein, die zur Entwurfszeit (als Ergebnis des Build-Prozesses) in die Assembly integriert werden. Eine Assembly kann mehrere Manifestressourcen enthalten. "Manifestressource" heißt es deshalb, weil die Namen der Ressourcen im Manifest der Assembly abgelegt sind. Zur Laufzeit kann jede Manifestressource über ihren Namen als Stream ausgelesen werden.

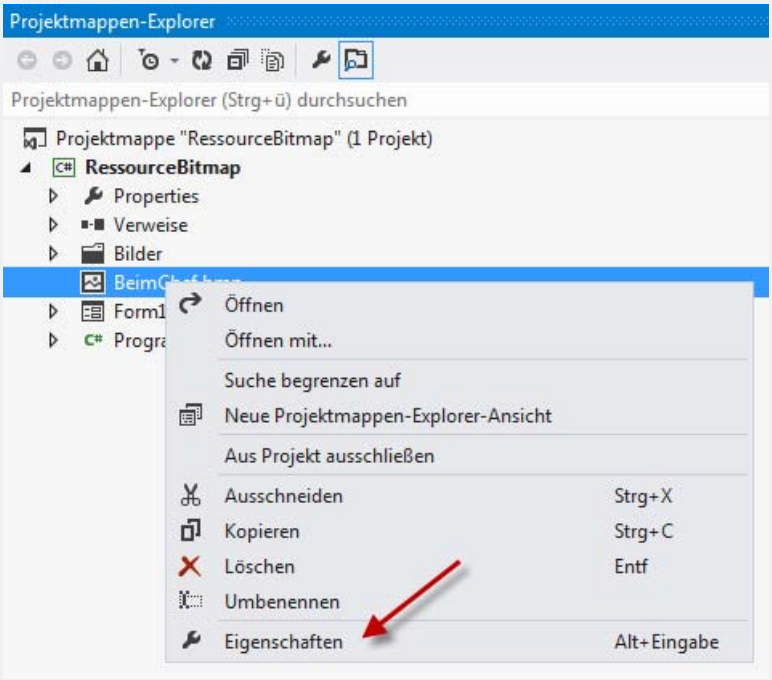
37.1.1 Erstellen von Manifestressourcen

Normalerweise verwendet man zum Anlegen einer Manifestressource den Assembly Linker (*al.exe*). Besitzer von Visual Studio können aber auf dieses Tool locker verzichten, da sie die entsprechenden Dateien lediglich zum Projekt hinzufügen und die *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource* setzen müssen.

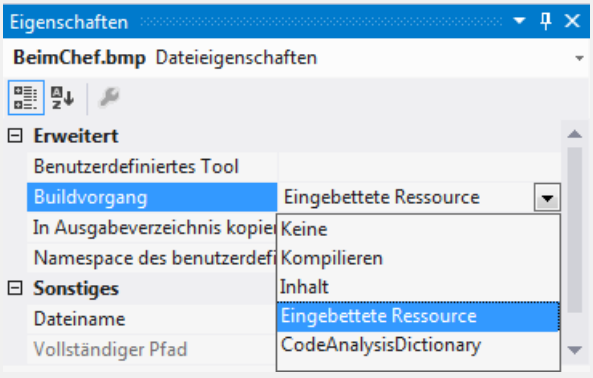
BEISPIEL 37.1: Manifestressource erstellen

C#

Um die Bilddatei *Bild1.jpg* als Manifestressource anzulegen, wählen Sie im Kontextmenü des Projektmappen-Explorers *Hinzufügen|Vorhandenes Element...* (oder Hauptmenü *Projekt|Vorhandenes Element hinzufügen...*) und selektieren die Datei *Bild1.jpg*. Im Kontextmenü von *Bild1.jpg* klicken Sie auf *Eigenschaften*:



Im Eigenschaftendialog stellen Sie die Eigenschaft *Buildvorgang* auf *Eingebettete Ressource*:



Nach dem Kompilieren der Anwendung befindet sich die Bilddatei in der erzeugten Assembly, Sie brauchen also keine weitere Datei mitzugeben.

37.1.2 Zugriff auf Manifestressourcen

Der übliche Weg zu eingebetteten Ressourcen führt über die Methoden der *Assembly*-Klasse. So kann man die *GetManifestResourceNames*- bzw. *GetManifestResourceStream*-Methode verwenden, um alle eingebetteten Ressourcen einer bestimmten Assembly aufzulisten bzw. zu laden. Bevor wir aber zu Einzelheiten kommen, wollen wir erläutern, nach welchem Muster die Namensvergabe erfolgt.

Namensgebung eingebetteter Ressourcen

Allgemein entspricht der Name einer eingebetteten Ressource folgendem Muster:

SYNTAX: `<DefaultNamespace>.<Unterverzeichnisse>.Dateiname`

Der *DefaultNamespace* wird in den Projekteigenschaften festgelegt und ist meist identisch mit dem Namen der Assembly (z.B. *WindowsApplication1*). Falls sich – wie in unserem Fall – die Ressource im Rootverzeichnis des Projekts befindet, entfallen die *Unterverzeichnisse* und der Name ist *<DefaultNamespace>.Dateiname*. Mehrere Unterverzeichnisse sind nicht durch Schrägstrich, sondern durch Punkte voneinander zu trennen. Es ist möglich, dass Sie spezielle Verzeichnisse innerhalb Ihres Projekts erzeugen, um dort die Ressource(n) abzulegen, z.B. ein Verzeichnis *\Bilder*. Dann wäre der Name der Ressource z.B. *WindowsApplication1.Bilder.Bild1.jpg*.

HINWEIS: Ist der Code einmal generiert, so können Sie zwar den Namespace nachträglich ändern, nicht aber die Unterverzeichnisse, die zur Benennung einer eingebetteten Ressource benutzt wurden!

Auflisten aller eingebetteten Ressourcen

Für diesen Zweck kommt die *GetManifestResourceNames*-Methode der *Assembly*-Klasse zur Anwendung.

BEISPIEL 37.2: Auflisten aller eingebetteten Ressourcen

C# Über einen Dateidialog wird eine Assembly geladen. Die Namen aller darin enthaltenen Ressourcen werden in einer *ListBox* angezeigt.

```
using System.Reflection;
using System.IO;
...

Assembly  ass = Assembly.LoadFrom(openFileDialog1.FileName);
string [] resNames = ass.GetManifestResourceNames();
listBox1.Items.Clear();
if( resNames.Length > 0 )
{
   listBox1.BeginUpdate();
   foreach(string resName in resNames)
   {
```

BEISPIEL 37.2: Auflisten aller eingebetteten Ressourcen

```
      listBox1.Items.Add(resName);
    }
    listBox1.EndUpdate();
}
```

Die Inhalte eingebetteter Ressourcen auslesen

Hierfür verwenden Sie die *GetManifesResourceStream*-Methode der *Assembly*-Klasse:

BEISPIEL 37.3: Eine in der eigenen Assembly eingebettete Bildressource wird angezeigt.

C#

```
...
using System.Reflection;
using System.IO;
...
Assembly ass = Assembly.GetExecutingAssembly();
if (pictureBox1.Image != null)
   pictureBox1.Image.Dispose();
Stream strm = ass.GetManifestResourceStream("RessourceBitmap.Bild1.jpg");
pictureBox1.Image = new Bitmap(strm);
```

Die *GetManifestResourceStream*-Methode hat zwei Überladungen. Die erste erwartet den Namen der Ressource, die zweite erwartet stattdessen einen Typ und einen String. Das vereinfacht die Namensgewinnung für die Ressource: Intern wird der Namespace des Typs genommen und mit dem String wird der Name vervollständigt.

Einige Klassen des .NET Framework verwenden dieses Verfahren ebenfalls, aber anstatt *Stream*-Objekte zurückzugeben erzeugen sie damit ein bestimmtes Objekt. So hat die *Bitmap*-Klasse einen Konstruktor, welcher eine eingebettete Ressource in ein *Bitmap*-Objekt laden kann.

BEISPIEL 37.4: Eine deutlich kürzere Version des obigen Beispiels.

C#

```
if (pictureBox1.Image != null)
     pictureBox1.Image.Dispose();
pictureBox1.Image = new Bitmap(typeof(Form1), "Bild1.jpg");
```

Dem *Bitmap*-Konstruktor wird auf diese Weise mitgeteilt, eine Ressource zu finden, deren *Namespace* dem von *Form1* entspricht (weil *Form1* im Rootverzeichnis des Projekts liegt).

Eine weitere Variante der zweiten Zeile:

```
pictureBox1.Image = new Bitmap(this.GetType(), "Bild1.jpg");
```

37.2 Typisierte Ressourcen

Typisierte Ressourcen bauen auf einfachen Manifestressourcen auf. Es handelt sich hierbei um Zusammenstellungen von Schlüssel-Wert-Paaren, wobei der Schlüssel ein eindeutiger String und der Wert ein beliebiges Objekt ist. Im Gegensatz zu den Manifestressourcen sind diese Ressourcen typisiert und wesentlich effizienter, da sie nicht streambasiert arbeiten. Die Bereitstellung erfolgt in der Regel als Datei mit der Extension *.resources*.

37.2.1 Erzeugen von .resources-Dateien

Dazu benötigen Sie einen *ResourceWriter* aus dem Namespace *System.Resources*.

BEISPIEL 37.5: Erzeugen einer *.resources*-Datei

C#

Die Methode *createResources* erzeugt eine Ressourcendatei, die Texte für beliebige Meldungen enthält.

```
static void createResources()
{
    ResourceWriter rw = new System.Resources.ResourceWriter("Messages.resources");
```

Die String-Ressourcen als Schlüssel-Wert-Paar hinzufügen:

```
    rw.AddResource("1", "Wahrscheinlich haben Sie recht.");
    rw.AddResource("2", "Ja, natürlich.");
    rw.AddResource("3", "Versuchen Sie es später noch einmal!");
    rw.AddResource("4", "Bitte keine Ablenkungsmanöver!");
    rw.AddResource("5", "Leider nein.");
    rw.AddResource("6", "Wie ich sehe - ja!");
```

Die Datei *Messages.resources* wird erzeugt und geschlossen:

```
    rw.Generate();
    rw.Close();
}
```

Nach Aufruf von *createResources()* werden Sie im Unterverzeichnis *\bin\Debug* die Datei *Messages.resources* vorfinden.

37.2.2 Hinzufügen der .resources-Datei zum Projekt

Die erzeugte Ressourcendatei *Messages.resources* fügen Sie – genauso wie oben für eine Manifestressource beschrieben – dem Projekt hinzu. Wählen Sie also das Menü *Projekt|Vorhandenes Element hinzufügen...* und setzen Sie die *Buildvorgang*-Eigenschaft auf *Eingebettete Ressource*. Alternativ können Sie auch das *Hinzufügen*-Kontextmenü im Projektmappen-Explorer verwenden.

HINWEIS: Da die Datei *Messages.ressources* nach dem Kompilieren in der Assembly *ResourcesTest.exe* eingebettet ist, können Sie sie auch aus dem Verzeichnis *\bin\Debug* löschen.

37.2.3 Zugriff auf die Inhalte von .resources-Dateien

Für den Zugriff spielt die Klasse *ResourceManager* (Namespace *System.Resources*) eine wichtige Rolle, sie benötigt zur Instanziierung zwei Parameter: einen Verweis auf die *resources*-Datei, die als Manifestressource in der Assembly abgelegt ist, sowie einen Verweis auf die Assembly selbst.

Der *ResourceManager* stellt für den Zugriff auf Ressourcen-Elemente die Methoden *GetString*, *GetObject* und *GetStream* bereit. Anzugeben ist der Name des Ressourcen-Elements unter Beachtung der Groß-/Kleinschreibung. Die Bedeutung der Groß-/Kleinschreibung kann aber mittels *IgnoreCase*-Eigenschaft deaktiviert werden.

BEISPIEL 37.6: Zugriff auf die Inhalte von .resources-Dateien

C#

Der Zugriff auf die im Projekt *ResourcesTest* eingebettete Datei *Messages.resources* erfolgt über die *GetString*-Methode des *ResourceManager*, wobei der Methode der Schlüssel als Parameter übergeben wird. Um einen einfachen Test zu ermöglichen, erzeugen wir den Schlüssel mit einem Zufallsgenerator, der zugehörige Wert wird in einem *Label* angezeigt.

```
using System.Resources;
...
private ResourceManager rm =  new ResourceManager("ResourcesTest.Messages",
                                        System.Reflection.Assembly.GetExecutingAssembly());
System.Random z = new System.Random();
string num = z.Next(1, 41).ToString();
label1.Text = rm.GetString(num);
```

Bemerkungen zum Zugriff auf .resources-Dateien

- Die Extension *resources* ist **kein** Bestandteil des Namens, der dem Konstruktor als Argument übergeben wird.
- Das Objekt für die Assembly, in der sich der laufende Code befindet, erhält man über die Methode *System.Reflection.Assembly.GetExecutingAssembly*.
- Die *GetString*-Methode des *ResourceManager* ist eine typsichere Alternative zur *GetObject*-Methode.

37.2.4 ResourceManager einer .resources-Datei erzeugen

Wollen Sie die Ressourcen nachträglich manipulieren, so ist die gezeigte Vorgehensweise nicht geeignet, da die *.resources*-Datei unmittelbar nach ihrem Erzeugen erst per Hand in das Projekt "eingebettet" werden muss und nach dem Kompilieren – wie jede andere Manifestressource auch – nicht nachträglich manipuliert werden kann.

Ein Ausweg bietet sich, wenn Sie die Datei nicht in die Assembly einbetten, sondern als separate Datei mitführen. Dann können Sie einen passenden *ResourceManager* mithilfe der statischen Methode *CreateFilebasedResourceManager* erzeugen.

BEISPIEL 37.7: Erzeugen eines *ResourceManager* aus der Datei *Messages.resources.*

C#
```
private ResourceManager rm = ResourceManager.CreateFileBasedResourceManager(
                                    "Messages", Application.StartupPath, null);
```

In diesem Fall befindet sich die Datei *Messages.resources* im gleichen Verzeichnis wie die Assembly.

Bemerkungen

- Das Pendant zum *ResourceWriter* ist der *ResourceReader*, mit dem sich – als Alternative zum gezielten Zugriff per *ResourceManager* – komplette Ressourcendateien verarbeiten lassen.
- Bequemer als per Code können Ressourcen auch mit dem in Visual Studio integrierten Ressourceneditor angelegt werden (Menü *Projekt|Neues Element hinzufügen...|Ressourcen-datei*) .

37.2.5 Was sind .resx-Dateien?

Ressourcen liegen in *resources*-Dateien in einem binären Format vor. Anstatt, wie oben gezeigt, mittels *RessourceWriter* lassen sie sich aber auch aus **.resx*-Dateien erzeugen.

HINWEIS: *resx*-Dateien sind XML-Dokumente!

Zur Umwandlung einer binären *resources*- in eine XML-basierte *resx*-Datei können Sie das Tool *resgen.exe* (*Resource File Generator*) einsetzen.

Visual Studio bietet einen komfortablen Editor für *resx*-Dateien. Der Aufruf erfolgt über den Menüpunkt *Projekt|Neues Element hinzufügen... Ressourcendatei* oder einfach durch Doppelklick auf eine vorhandene *resx*-Datei im Projektmappen-Explorer.

HINWEIS: *resx*-Ressourcendateien haben besondere Bedeutung im Zusammenhang mit der Lokalisierung von Anwendungen, siehe ab Seite 1825!

37.3 Streng typisierte Ressourcen

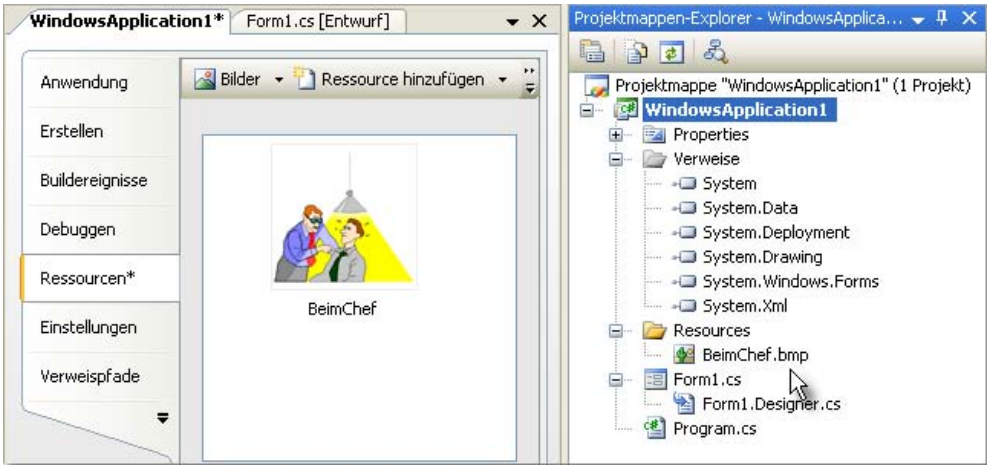
Beim Arbeiten mit dem *ResourceManager*-Objekt kann es häufig zu Fehlern kommen, da die Namen der Ressourcenelemente als Zeichenketten vorliegen und Tippfehler dazu führen, dass das Element nicht gefunden wird. Aus diesem Grund werden im .NET-Framework auch streng typisierte Ressourcen (*strongly-typed resources*) unterstützt.

So bietet das Tool *resgen.exe* die neue Option, eine Wrapper-Klasse für eine beliebige Ressource-Datei zu generieren, sodass der Programmierer mit *early binding* auf die Ressourcennamen zugreifen kann. Laufzeitfehler aufgrund falscher Ressourcennamen lassen sich so vermeiden.

37.3.1 Erzeugen streng typisierter Ressourcen

Visual Studio stellt einen Designer bereit, der unter anderem auch das automatische Generieren von Wrapper-Klassen für Ressourcen übernimmt. Die Bedienung ist sehr einfach:

Über das Menü *Projekt|...-Eigenschaften...* (oder durch Doppelklick auf den *Properties*-Eintrag im Projektmappen-Explorer) öffnen Sie die Seite "Ressourcen" des Projektdesigners.



Im Ergebnis wurde innerhalb des Projektverzeichnisses ein neues Unterverzeichnis namens */Resources* angelegt, in welchem die Bitmap gespeichert ist.

37.3.2 Verwenden streng typisierter Ressourcen

Die Verwendung der streng typisierten Ressourcen ist recht simpel, der Zugriff erfolgt über das *Properties.Resources*-Objekt.

BEISPIEL 37.8: Anzeige einer Bitmap

C#

```
pictureBox1.Image = WindowsApplication1.Properties.Resources.BeimChef;
```

Da auch eine Unterstützung per Intellisense erfolgt, wird die Fehleranfälligkeit deutlich verringert:

37.3.3 Streng typisierte Ressourcen per Reflection auslesen

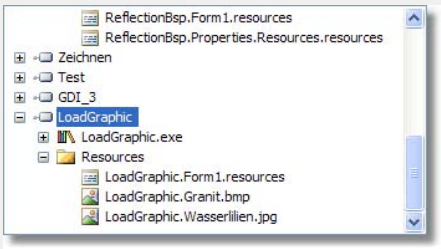
Für den Zugriff auf Ressourcen wie Grafiken, Videos, Sound etc. bietet sich die *GetManifestResourceStream*-Methode an.

Ab Version 2.0 des Frameworks hat sich aber hier eine kleine Änderung eingeschlichen, wie eine Kontrolle mit dem .NET-Refector zeigt:

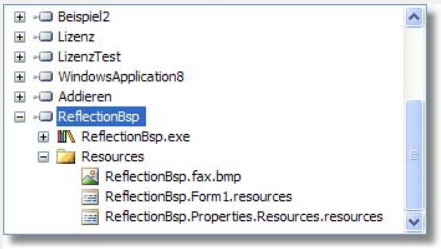
BEISPIEL 37.9: Unterschiede je nach .NET-Version

C#

.NET 1.x, (Eingelagerte Ressourcen werden direkt unter *Resources* gespeichert, z.B. *Wasserlilien.jpg*):



Ab .NET 2.x (das Verhalten für eingelagerte Ressourcen hat sich zwar nicht geändert, Ressourcen, die über die Projekt-Eigenschaften hinzugefügt wurden, sind aber jetzt im Stream *...Properties.Resources.resources* gespeichert):



Wir müssen uns deshalb auch beim Auslesen der Ressourcen etwas mehr bemühen als bisher.

BEISPIEL 37.10: Alle Ressourcen ermitteln

C#

```
...
using System.Reflection;
using System.IO;
using System.Resources;
using System.Collections;
```

BEISPIEL 37.10: Alle Ressourcen ermitteln

Assembly laden:

```
myass = Assembly.GetExecutingAssembly();
foreach (String s in myass.GetManifestResourceNames())
{
```

Hier bestimmen wir zunächst die einzelnen Ressource-Streams:

```
    listBox1.Items.Add(s);
    if (s.ToLower().EndsWith(".resources"))
```

Wenn in diesem *Stream* weitere Ressourcen enthalten sind:

```
    {
      Stream stream = myass.GetManifestResourceStream(s);
      ResourceReader Reader = new ResourceReader(stream);
      IDictionaryEnumerator id = Reader.GetEnumerator();
      while (id.MoveNext())
      {
```

ID.key bezeichnet die gleiche Ressource, die Sie auch mit *Properties.Resources.xyz* auslesen können:

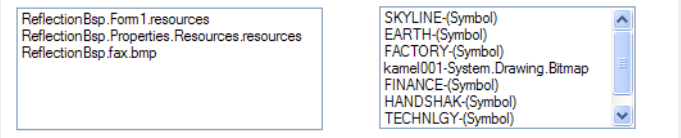
```
        listBox2.Items.Add(id.Key + "-" + id.Value);
```

Über *id.Value* können wir direkt auf die einzelnen Bitmaps zugreifen:

```
        if (id.Value is Bitmap)
        {
            Bitmap bmp = (Bitmap) (id.Value as Bitmap).Clone();
            bmp.Save(id.Key + ".bmp");
        }
      }
      Reader.Close();
    }
}
```

Ergebnis

Die Anzeige in *listBox1* und *listBox2*:



HINWEIS: Nach dem Ausführen obigen Beispiels (siehe Buch-Beispieldaten), werden alle Bitmaps aus der Assembly extrahiert.

BEISPIEL 37.11: Auslesen der eingebetteten Ressource *fax.bmp* und speichern in einer externen Datei bei Doppelklick auf den entsprechenden Eintrag in *listBox1*

C#
```
private void listBox1_MouseDoubleClick(object sender, MouseEventArgs e)
{
    Bitmap  bmp = new Bitmap(
    Assembly.GetExecutingAssembly().GetManifestResourceStream((String)
                                        listBox1.SelectedItem));
    bmp.Save((string) listBox1.SelectedItem);
}
```

HINWEIS: Achten Sie beim Zugriff auf die Ressourcen peinlichst auf die korrekte Schreibweise (Groß-/Kleinschreibung!).

37.4 Anwendungen lokalisieren

Das .NET Framework ermöglicht die komfortable Lokalisierung von Anwendungen. Texte und andere sprachabhängige Informationen befinden sich nicht mehr im eigentlichen Quellcode, sondern werden in eigenen Assemblies (so genannte Satelliten-Assemblies) bereitgestellt, die parallel zur eigentlichen Assembly existieren, darüber hinaus jedoch keinen Code beinhalten.

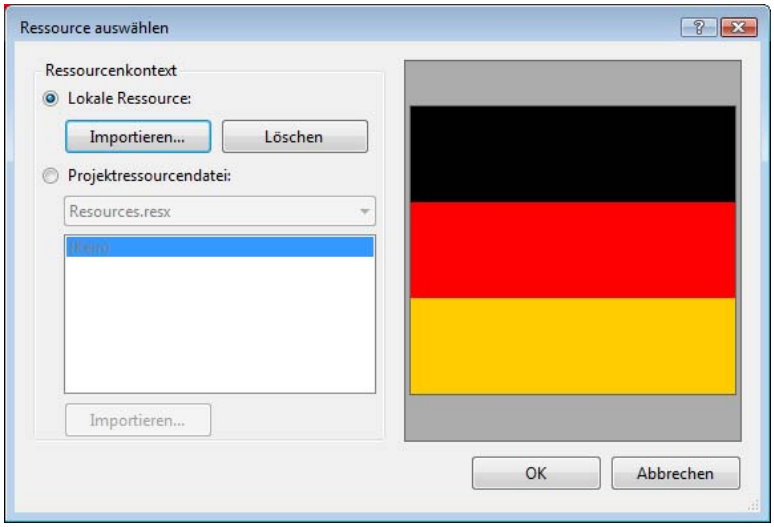
37.4.1 Localizable und Language

Visual Studio kann automatisch die Ressourcendateien für die zu lokalisierenden Elemente der Benutzerschnittstelle, wie z.B. Beschriftungen der Steuerelemente eines Formulars, erzeugen. Von Bedeutung sind dabei die Formulareigenschaften *Localizable* und *Language*.

HINWEIS: *Localizable* und *Language* sind "künstliche" Formulareigenschaften, sie sorgen lediglich dafür, dass der Designer bei der Code-Generierung die zu lokalisierenden Informationen in der richtigen *.resx*-Datei ablegt.

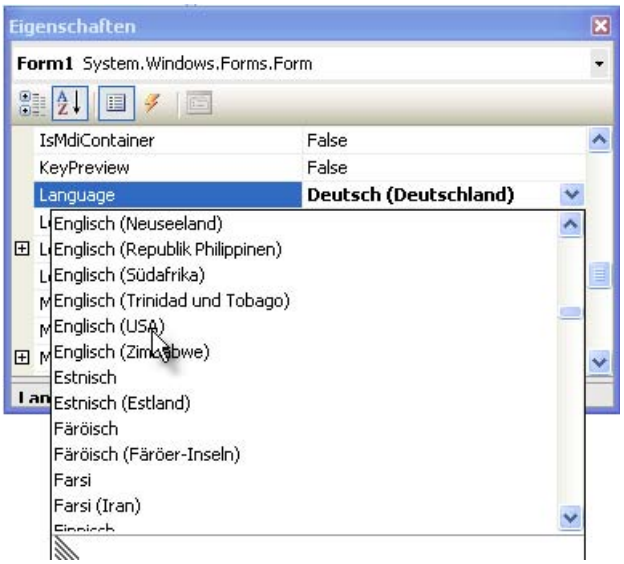
37.4.2 Beispiel "Landesfahnen"

In einer neuen Windows-Anwendung setzen wir die *Localizable*-Eigenschaft von *Form1* auf *True*, die *Language*-Eigenschaft verbleibt zunächst auf ihrem Standardwert (*Default*). Auf *Form1* befinden sich ein *Label*, eine *PictureBox* und ein *Button*. Der *Image*-Eigenschaft der *PictureBox* weisen wir per Dialog eine Bitmap mit der deutschen Flagge zu:



Die Beschriftung erfolgt in deutscher Sprache, sodass die Standardversion etwa so aussieht:

Nun stellen wir im Eigenschaftenfenster von *Form1* die *Language*-Eigenschaft auf *Deutsch* ein. Dazu selektieren wir unser Land aus einer schier endlos langen Liste, in welcher (fast) jedes Land der Erde vertreten ist.



Wählen Sie im Anschluss *Englisch (USA)* als neue *Language* und gestalten Sie auf analoge Weise die US-Version von *Form1*:

Ein Blick in den Projektmappen-Explorer zeigt, dass die Ressourcen-Dateien *Form1.de.resx* und *Form1.en-US.resx*, welche die sprachabhängigen Informationen kapseln, hinzugekommen sind.

Wie z.B. ein Doppelklick auf *Form1.en-US.resx* zeigt, sind die Text-Ressourcen als Schlüssel-Wertepaar hinterlegt:

Nach dem Kompilieren werden wir zunächst nur die deutsche Version des Programms zu Gesicht bekommen, es sei denn, wir ändern die Spracheinstellung des aktuellen Threads. Dazu fügen wir die fett gedruckten Programmzeilen hinzu:

HINWEIS: Nach wie vor haben wir es mit einem einzigen Formular (*Form1*) zu tun, es ist deshalb völlig egal, von welcher Formularansicht aus wir das Quellcodefenster öffnen.

```
using System.Globalization;   // !
using System.Threading;       // !
...
        public Form1()
        {
```

Wichtig ist, dass die Einstellung der *CurrentUICulture* **vor** dem Aufruf von *InitializeComponent()* erfolgt:

```
            // Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");
             Thread.CurrentThread.CurrentUICulture = new CultureInfo("en-US");
             InitializeComponent();
        }
    }
}
```

37.4.3 Einstellen der aktuellen Kultur zur Laufzeit

Damit zum Ändern der *CurrentUICulture* nicht immer das Programm neu kompiliert werden muss, empfiehlt sich für Demozwecke das Vorschalten eines weiteren Formulars *Form2*, in welchem die gewünschte Sprache zur Laufzeit gewechselt werden kann:

Der Code des "Vorschaltformulars" *Form2*:

```
private void button1_Click(object sender, EventArgs e)
{
    string ci = "de-DE";
    if (radioButton2.Checked) ci = "en-US";
    Form1 frm = new Form1(ci);
    frm.Show();
}
```

Den Konstruktor von *Form1* ändern wir wie folgt:

```
public Form1(string ci)
{
     Thread.CurrentThread.CurrentUICulture = new CultureInfo(ci);
     InitializeComponent();
}
```

Schließlich muss das Startformular der Anwendung im Hauptprogramm *Program.cs* auf *Form2* gesetzt werden:

```
static class Program
{
...
        Application.EnableVisualStyles();
        Application.Run(new Form2());
     }
}
```

Bemerkungen

- Neben der Hauptassembly wurden zwei neue Ordner (*\de* und *\en-US*) erzeugt, die jeweils eine Satelliten-Assembly **.resources.dll* enthalten (Satelliten-Assemblies bestehen nur aus Ressourcen).
- Die Original-Bilddateien der deutschen und der amerikanischen Flagge sind nur für den Entwurf des Projekts, nicht aber für die Weitergabe der kompilierten Anwendung erforderlich, da auch die kompletten Bildressourcen in den Satelliten-Assemblies eingelagert sind.

- Die Lokalisierung von .NET-Anwendungen beschränkt sich nicht nur auf das Anlegen von Text-Ressourcen. Nahezu jede Eigenschaft, wie z.B. die Höhe oder Breite eines jeden Formulars oder Steuerelements, lässt sich landesspezifisch anpassen
- Zwar ist es auch möglich, Grafiken und andere Dateien direkt als Ressourcen in eine Assembly einzubetten oder mit der Assembly zu verlinken, allerdings unterstützen diese Ressourcen nicht die Lokalisierung, denn dazu müssen Grafiken etc. in eine *.resx*-Datei integriert werden.
- Die Spracheinstellung des aktuellen Threads richtet sich beim Programmstart nach den Spracheinstellungen von Windows. Zur Laufzeit können Sie die Sprache wechseln, z.B. mit:

```
Thread.CurrentThread.CurrentUICulture = new System.Globalization.CultureInfo("en-GB");
```

- Die aktuelle Sprache ermitteln Sie mit

```
System.Threading.Thread.CurrentThread.CurrentUICulture.Name;
```

oder

```
System.Threading.Thread.CurrentThread.CurrentUICulture.NativName;
```

Name liefert den englischen Sprachnamen, *NativName* den Namen der Sprache in der aktuellen Sprache.

Kapitel 38

Komponentenentwicklung

Die komponentenbasierte Entwicklung gehört mit den Grundpfeilern der .NET-Philosophie. Visual Studio bietet vielfältige Features, die dem Programmierer die Entwicklung eigener Steuerelemente (Komponenten) erleichtern. Was Sie dabei erleben ist OOP pur, und wir setzen für die Lektüre dieses Kapitels voraus, dass Sie einigermaßen sattelfest in Begriffen wie Klassen, Vererbung, Konstruktor usw. sind (siehe Kapitel 3).

HINWEIS: Zur Programmierung von Webserver-Steuerelementen siehe Kapitel 42!

38.1 Überblick

Bevor Sie sich auf die Komponentenprogrammierung stürzen, sollten Sie sich gut überlegen, welchen Komponententyp Sie wirklich benötigen, bietet Ihnen doch Visual Studio gleich ein ganzes Arsenal von Möglichkeiten an:

Typ	Bemerkung/Verwendung
Benutzersteuerelement	Sie möchten ein oder mehrere Controls in einem Container zusammenfassen und mit einer neuen Schnittstelle/Funktionalität ausstatten. Ableitung von *UserControl*.
Benutzerdefiniertes Steuerelement	Sie wollen ein neues Control erstellen oder ein vorhandenes um zusätzliche Funktionen erweitern. Ableitung vom Urtyp *Control* bzw. von vorhandenen Steuerelementen.
Geerbtes Benutzersteuerelement	Sie möchten ein Benutzersteuerelement aus Ihrem oder aus anderen Projekten um weitere Funktionen/Controls erweitern.
Komponentenklasse	Sie wollen ganz weit unten anfangen und sich sowohl um Schnittstelle als auch Funktionalität komplett selbst kümmern. Ableitung von *Component*.

Unter ähnlichen und leicht verwechselbaren Namen (dem Übersetzer sei Dank!) verbergen sich teilweise grundverschiedene Ansätze. Im Folgenden wollen wir Ihnen deshalb zunächst die Grundkonzepte und Unterschiede an kleineren Beispielen vorstellen, bevor wir die Gemeinsamkeiten bei der Definition von Eigenschaften und Methoden bzw. beim Auslösen von Ereignissen besprechen.

Im Anschluss beschäftigen wir uns noch mit einigen interessanten Fragen im Zusammenhang mit der Komponentenentwicklung.

38.2 Benutzersteuerelement

Hierbei handelt es sich quasi um einen Container für beliebige Steuerelemente, die damit zu einer Einheit verschmolzen werden können. Die Entwicklung erfolgt (wie bei einer normalen Windows-Anwendung) rein visuell, Sie platzieren die konstituierenden Steuerelemente im Container und legen deren Eigenschaften fest. In einem weiteren Schritt können Sie Ihrem Benutzersteuerelement ein eigenes Interface mit Eigenschaften, Methoden und Ereignissen geben.

Damit dürften sich auch schon die Vor- und Nachteile dieses Typs klar herausstellen:

- Die Entwicklung von Benutzersteuerelementen ist, dank visueller Unterstützung, recht einfach und schnell möglich.
- Die Programmierlogik zwischen den enthaltenen Controls kann einfach realisiert werden und ist in der Gesamtkomponente gekapselt.
- Eigenschaften und Methoden der enthaltenen Komponenten können sicher vor dem Anwender des Benutzersteuerelements ausgeblendet werden.
- Nachteilig ist der teilweise erhebliche Aufwand für das Erstellen einer sinnvollen Programmierschnittstelle (Eigenschaften/Methoden/Ereignisse).

Damit dürfte sich dieser Komponententyp hauptsächlich für Routine-Programmieraufgaben anbieten, bei denen eine öfter wiederkehrende Logik in einer Komponente gekapselt werden soll.

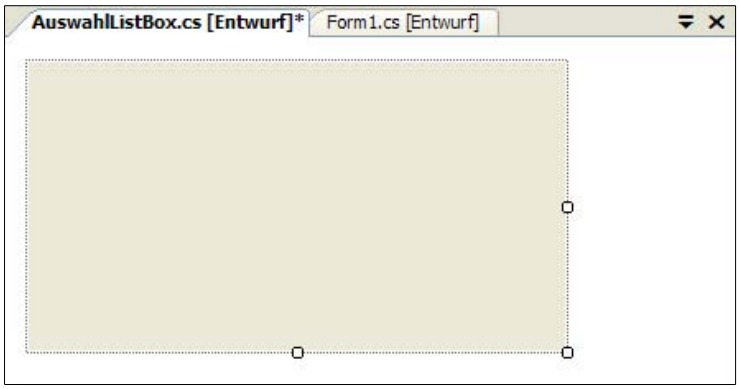
38.2.1 Entwickeln einer Auswahl-ListBox

HINWEIS: Um Ihnen den Test der neuen Komponenten möglichst einfach zu machen, integrieren wir die Komponente in ein normales Windows-Projekt. Im Normalfall werden Sie die Komponente sicher in einer "Windows-Steuerelemente-Bibliothek" unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

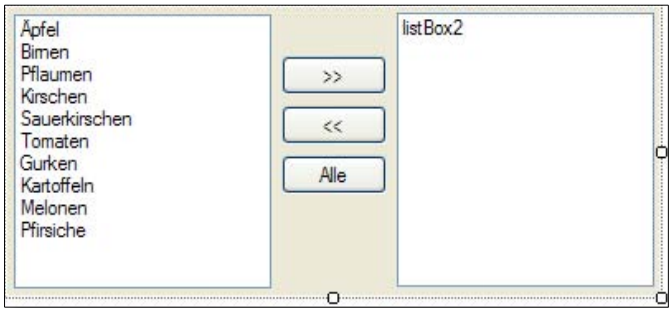
Erstellen Sie zunächst eine neue Windows Forms-Anwendung und fügen Sie über den Menüpunkt *Projekt|Benutzersteuerelement hinzufügen* ein neues Benutzersteuerelement unter dem Namen *AuswahlListBox.cs* hinzu.

Oberflächendesign

Im Designer finden Sie jetzt bereits den "Container" für die zu platzierenden Steuerelemente vor:



Die Optik und das Handling dürften Ihnen vom normalen Formularentwurf her bereits bekannt vorkommen, platzieren Sie einfach zwei *ListBox*en und drei *Button*s innerhalb des obigen Steuerelements:



Der linken *ListBox* (*listBox1*) fügen Sie im Eigenschafteneditor einige Einträge (über die *Items*-Eigenschaft) hinzu.

Implementieren der Programmlogik

Jetzt noch schnell etwas Code hinzufügen, und fertig ist die neue Komponente.

Löschen der bisherigen Einträge und Kopieren **aller** Einträge in *listBox2*:

```
private void button3_Click(object sender, EventArgs e)
{
    listBox2.Items.Clear();
    listBox2.Items.AddRange(listBox1.Items);
}
```

Verschieben eines Eintrags von *listBox2* in *listBox1*:

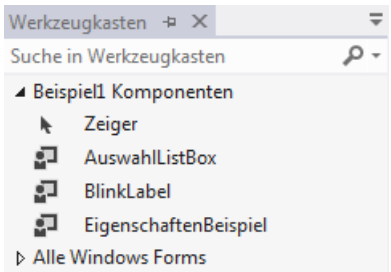
```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        listBox1.Items.Add(listBox2.Items[listBox2.SelectedIndex]);
        listBox2.Items.RemoveAt(listBox2.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

Verschieben eines Eintrags von *listBox1* in *listBox2*:

```
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        listBox2.Items.Add(listBox1.Items[listBox1.SelectedIndex]);
        listBox1.Items.RemoveAt(listBox1.SelectedIndex);
    }
    catch (Exception)
    {
        MessageBox.Show("Keine Auswahl getroffen!");
    }
}
```

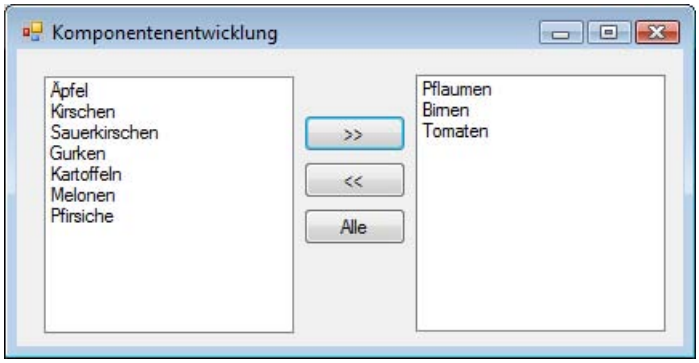
38.2.2 Komponente verwenden

Zunächst kompilieren Sie die aktuelle Anwendung. Ein Blick in den Werkzeugkasten sollte im Erfolgsfall bereits die neue Komponente zeigen:



Mit dem bereits im Projekt vorhandenen Formular können wir uns jetzt an einen ersten Test wagen. Fügen Sie die neue Komponente ein und starten Sie das Programm. Bereits jetzt verfügt Ihre Komponente über jede Menge Funktionalität, was fehlt ist jedoch eine Interaktion mit dem eigentlichen Programm. Zu diesem Zweck können Sie weitere Eigenschaften einführen, um zum Beispiel die

Einträge in *listBox2* abzufragen oder die Einträge von *listBox1* vorzudefinieren. Doch dazu später mehr.



38.3 Benutzerdefiniertes Steuerelement

Ein benutzerdefiniertes Steuerelement verwenden Sie, wenn Sie von vorhandenen Controls (z.B. *TextBox*, *Label*, *Timer* etc.) bzw. vom Urtyp *Control* eine neue Komponente **ableiten** wollen. Ihr Steuerelement erbt zunächst alle Eigenschaften, Ereignisse und Methoden des Vorgängers.

Sie können darauf aufbauend

- eigene Eigenschaften,
- Methoden und
- Ereignisse implementieren sowie
- Methoden und Ereignisse ausblenden.

HINWEIS: Die äußeren Abmessungen der Komponente sind zunächst durch den Vorfahren bestimmt, um ein Zeichnen der Komponente mittels *Paint* brauchen Sie sich nicht zu kümmern, solang Sie nicht *Control* als Vorfahren verwenden.

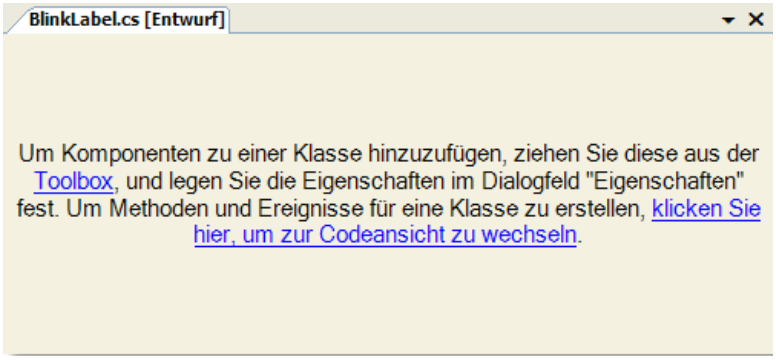
38.3.1 Entwickeln eines BlinkLabels

HINWEIS: Um Ihnen den Test der neuen Komponenten möglichst einfach zu gestalten, integrieren wir die Komponente in ein normales Windows-Projekt. Im Normalfall werden Sie die Komponente in einer "Windows-Steuerelemente-Bibliothek" unterbringen, da nur so eine einfache Wiederverwendbarkeit gegeben ist.

Wie beim Benutzersteuerelement erstellen wir zunächst eine neue Windows Forms-Anwendung und fügen über den Menüpunkt *Projekt|Neues Element hinzufügen* ein neues *Benutzerdefiniertes Steuerelement* unter dem Namen *BlinkLabel.cs* hinzu.

Oberflächendesign

Zum Projekt wurde automatisch die folgende Ansicht hinzugefügt:



Nicht sehr viel Ähnlichkeit mit einem *Label*, werden Sie sicher bemerken, aber hier handelt es sich lediglich um einen Dummy, der für alle Klassen bzw. Vorfahren gleich aussieht.

Festlegen des Typs des Vorfahren

Ein Blick in die Liste der Eigenschaften zeigt bereits jetzt jede Menge Properties. Doch ach, bisher wird *System.Windows.Forms.Control* als Klassentyp angezeigt, was auch richtig ist, da wir noch keinen eigenen Vorfahrtyp bestimmt haben. Das wollen wir nun nachholen, indem wir in die Code-Ansicht wechseln (Doppelklick auf den Dummy).

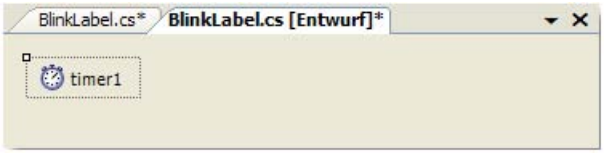
Hier suchen Sie die Klassendeklaration

```
using System;
...

namespace Beispiel1
{
    public partial class BlinkLabel : Control
    {
...
```

... und ersetzen sie durch die folgende Zeile:

```
...
    public partial class BlinkLabel : Label
...
```

Bei einem Blick in die Eigenschaftenliste werden Sie alle Label-spezifischen Eigenschaften und Ereignisse vorfinden. Wechseln Sie nun wieder zurück zum Dummy und fügen Sie einen *Timer* ein. Das Ganze sieht im Moment zwar etwas merkwürdig aus, aber es funktioniert.



HINWEIS: Sie können den *Timer* natürlich auch per Code erzeugen, was sicher eleganter ist, doch wir bohren diesmal das Brett an der dünnsten Stelle.

Implementieren der Programmlogik

Nach einem Doppelklick auf den *Timer* und dem Wechsel in die Codeansicht dürfte sich Ihnen der folgende Anblick bieten:

```
using System;
...
namespace Beispiel1
{
    public partial class BlinkLabel : Label
    {
        public BlinkLabel()
        {
            InitializeComponent();
            timer1.Interval = 500;
            timer1.Start();
        }

        protected override void OnPaint(PaintEventArgs pe)
        {
            // TODO: Benutzerdefinierten Zeichnungscode hier einfügen
            // OnPaint-Basisklasse wird aufgerufen
            base.OnPaint(pe);
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            this.Visible = !this.Visible;
        }
    }
}
```

Fügen Sie lediglich die fett hervorgehobenen Codezeilen ein, um das Steuerelement über den Konstruktor zu initialisieren und mittels *Timer* ein- und auszublenden. Das war es auch schon, und wir können uns nun um das Einbinden des Steuerelements kümmern.

Kompilieren Sie jedoch zunächst die Anwendung, um das neue Steuerelement in die Toolbox aufzunehmen.

38.3.2 Verwenden der Komponente

Öffnen Sie das zum Projekt gehörende Formular *Form1* und platzieren Sie das *BlinkLabel* auf dem Formular, um sich von der Funktionstüchtigkeit zu überzeugen.

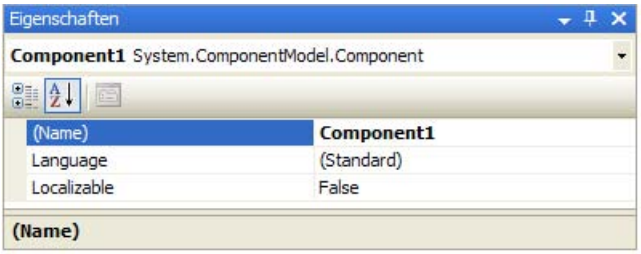
Bereits im Entwurfsmodus beginnt das Label nervend zu blinken!

HINWEIS: Natürlich handelt es sich hier nur um ein ziemlich simples Steuerelement ohne weitere Eigenschaften. Haben Sie sich aber durch dieses Kapitel durchgekämpft, sollte es Ihnen möglich sein, zum Beispiel die Blink-Frequenz über eine eigene Eigenschaft festzulegen oder weitere Ereignisse an die Komponente zu binden.

38.4 Komponentenklasse

Möchten Sie ganz unten anfangen und lediglich die rudimentärsten Funktionen übernehmen, verwenden Sie eine Komponentenklasse.

Die vorgegebene Basis-Klasse *System.ComponentModel.Component* können Sie übernehmen. Bei einem Blick auf die Eigenschaftenliste werden Sie aber feststellen, dass es sich um einen absoluten Basistyp handelt, der erst mit Leben erfüllt werden muss:



Dieser Steuerelemente-Typ eignet sich zum Beispiel für den Entwurf nicht sichtbarer Komponenten (Multimedia-Timer, Datenbank-Objekte etc.).

Doch was unterscheidet eine Komponentenklasse eigentlich von einer ganz trivialen Klasse?

- Die Fähigkeit, im Designer angezeigt zu werden und Eigenschaften/Ereignisse im Eigenschaften-Fenster zu präsentieren, dürfte schnell erkannt sein.
- Ein Blick in den Quellcode zeigt uns jedoch auch, dass die Komponente in der Lage ist, weitere Komponenten aufzunehmen (*IContainer*):

```
namespace Beispiel1
{
    partial class Component1
    {
        private System.ComponentModel.IContainer components = null;
...
```

Damit können Sie auch hier per Drag&Drop zum Beispiel einen *Timer* oder ein Data Control einfügen, deren Eigenschaften konfigurieren und Ereignisse programmieren.

Ob Sie sich nun für das Erstellen einer Klasse oder einer Komponentenklasse entscheiden, hängt nur davon ab, wie viel Komfort Sie dem Endanwender bieten wollen.

HINWEIS: Auf ein eigenes Beispiel verzichten wir an dieser Stelle, da der prinzipielle Ablauf der Vorgehensweise beim *Custom Control* entspricht.

38.5 Eigenschaften

Im Folgenden wollen wir uns mit den verschiedenen Varianten und Optionen von Eigenschaften näher befassen.

HINWEIS: Die Ausführungen lassen sich auf alle der eingangs genannten drei Steuerelementetypen anwenden, auch wenn wir uns in den folgenden Beispielen auf benutzerdefinierte Steuerelemente beschränken werden.

38.5.1 Einfache Eigenschaften

Unter dieser Rubrik wollen wir Eigenschaften verstehen, die auf einfachen Basistypen (zum Beispiel *String* oder *Integer*) basieren. Im Eigenschaften-Fenster haben Sie die Möglichkeit, den Wert zu editieren (nur wenn die Eigenschaft dies auch zulässt).

Folgende Steuerungsmöglichkeiten und Optionen für die Eingabe bestehen:

- Nur Lesezugriff
- Schreib-/Lesezugriff
- Schreibzugriff
- Ausblenden im Eigenschaftenfenster
- Wertebereichsbeschränkung und Fehlerprüfung
- Hinzufügen von Beschreibungen
- Default-Werte
- Einfügen in Kategorien

38.5.2 Schreib-/Lesezugriff (Get/Set)

Hierbei dürfte es sich um die wohl am häufigsten verwendete Variante bei Eigenschaften handeln. Der Nutzer des Steuerelements hat die Möglichkeit, sowohl Eigenschaftswerte zu lesen als auch zu ändern. Dazu müssen Sie als Entwickler sowohl die *get*- als auch die *set*-Option implementieren.

BEISPIEL 38.1: Schreib-/Lesezugriff (mit Zugriffsmethoden)

C#

Die Variable für die interne Zustandsverwaltung:

```
private int _MeineIntegerEigenschaft;
```

Die beiden Zugriffsmethoden:

```
public int MeineIntegerEigenschaft
{
    get { return _MeineIntegerEigenschaft; }
    set { _MeineIntegerEigenschaft = value; }
}
```

38.5.3 Nur Lese-Eigenschaft (ReadOnly)

Möchten Sie dem Anwender lediglich einen Lesezugriff auf den Eigenschaftswert gestatten, lassen Sie bei der Deklaration der Eigenschaft einfach die *set*-Option weg.

BEISPIEL 38.2: Nur Lese-Eigenschaft

C#

Die Variable für die interne Zustandsverwaltung:

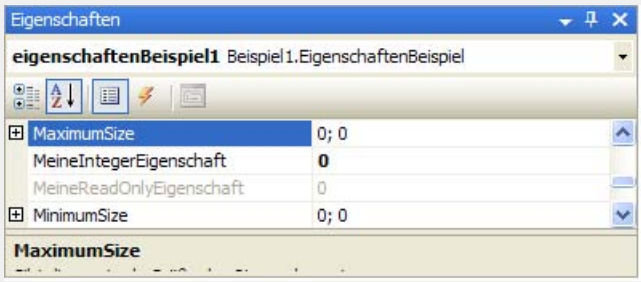
```
private int _MeineReadOnlyEigenschaft;
```

Die Zugriffsmethode:

```
public int MeineReadOnlyEigenschaft
{
    get { return _MeineReadOnlyEigenschaft; }
}
```

Ergebnis

Bei Verwendung der Komponente wird die Eigenschaft im Eigenschaftenfenster in grauer Schrift angezeigt, da sie schreibgeschützt ist.



38.5.4 Nur-Schreibzugriff (WriteOnly)

Bei dieser Variante wird die *get*-Option weggelassen.

```
private int _MeineWriteOnlyEigenschaft;
public int MeineWriteOnlyEigenschaft
{
    set { _MeineWriteOnlyEigenschaft = value; }
}
```

Allerdings sollten Sie für derartige Anwendungsfälle besser eine Methode verwenden, da dies den Sinn der Operation besser verdeutlicht.

HINWEIS: Die Eigenschaft wird sinnigerweise nicht im Eigenschaftenfenster angezeigt, welcher Wert sollte auch dargestellt werden?

38.5.5 Hinzufügen von Beschreibungen

Mit dem Attribut *Description* steuern Sie den Inhalt des Beschreibungsfeldes im Eigenschaftenfenster.

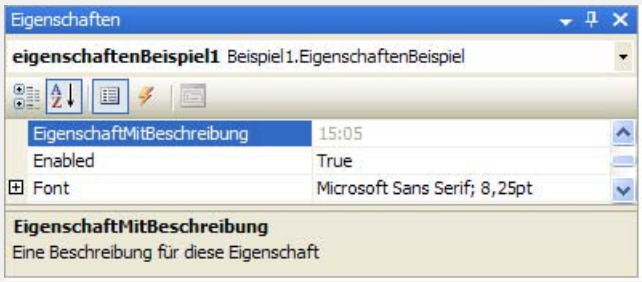
BEISPIEL 38.3: Eine Beschreibung festlegen

C#

```
[Description("Eine Beschreibung für diese Eigenschaft")]
public String EigenschaftMitBeschreibung
{
    get { return System.DateTime.Now.ToShortTimeString(); }
}
```

Ergebnis

Die Eigenschaft im Eigenschaftenfenster:



38.5.6 Ausblenden im Eigenschaftenfenster

Nicht in jedem Fall möchte man, dass eine Eigenschaft schon zur Entwurfszeit im Eigenschaftenfenster angezeigt wird. Über das Attribut *Browsable* haben Sie die Möglichkeit, die Sichtbarkeit der Eigenschaft zu steuern.

BEISPIEL 38.4: Einfügen eines Attributs

C#
```
[Browsable(false)]
 public int NichtSichtbareEigenschaft
 {
   get { return myValue; }
   set { myValue = value; }
 }
```

38.5.7 Einfügen in Kategorien

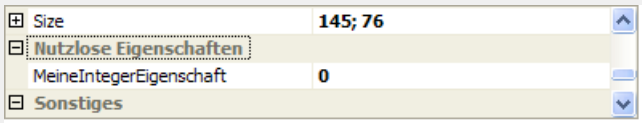
Auch hier dient ein Attribut (*Category*) dazu, den Eigenschaften weitere Informationen für den Eigenschafteneditor mit auf den Weg zu geben.

BEISPIEL 38.5: Einfügen von *MeineIntegerEigenschaft* in die Kategorie "Nutzlose Eigenschaften"

C#
```
[Category("Nutzlose Eigenschaften")]
public int MeineIntegerEigenschaft
{
    get { return _MeineIntegerEigenschaft; }
    set { _MeineIntegerEigenschaft = value; }
}
```

Ergebnis

Das Resultat im Eigenschaftenfenster:



HINWEIS: Dieses Attribut wirkt sich natürlich nur aus, wenn die Eigenschaften auch in Kategorien angezeigt werden.

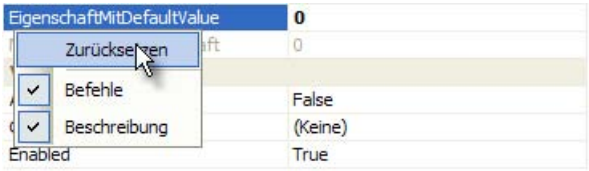
38.5.8 Default-Wert einstellen

Mit Hilfe des Attributs *DefaultValue* können Sie dem Anwender die Möglichkeit geben, über Reset bzw. Zurücksetzen der Eigenschaft einen vorgegebenen Wert zuzuweisen.

BEISPIEL 38.6: Standardwert zuweisen

C#
```
private int _myint;
[DefaultValue(55)]
public int EigenschaftMitDefaultValue
{
    get { return _myint; }
    set { _myint = value; }
}
```

Im Eigenschaftenfenster erreichen Sie die gewünschte Funktion über das Kontextmenü:



38.5.9 Standard-Eigenschaft (Indexer)

Da Programmierer so wenig wie möglich schreiben wollen, wurden für den Zugriff auf Array-Eigenschaften die Indexer erfunden.

Der Vorteil: Sie können auf die Angabe eines Eigenschaftennamens verzichten und direkt mit dem Objekt arbeiten.

BEISPIEL 38.7: Standardeigenschaft definieren

C#

```
    public partial class EigenschaftenBeispiel : Control
    {
...
        private string []mydata = {"rot","gelb","grün"};

        public string this[int index]
        {
            get { return mydata[index]; }
        }
```

Verwenden können Sie die Klasse später wie folgt:

```
        private void button1_Click(object sender, EventArgs e)
        {
            Text = eigenschaftenBeispiel1[0];
        }
```

38.5.10 Wertebereichsbeschränkung und Fehlerprüfung

Hierbei handelt es sich um eine der wohl komplexesten Aufgaben des Programmierers. Es geht darum, Fehleingaben des Anwenders zu verhindern bzw. die Eingabe auf gewünschte Werte zu beschränken. Dazu stehen dem Entwickler innerhalb der *set*-Zugriffsmethode alle Möglichkeiten offen.

BEISPIEL 38.8: Wertebereichsbeschränkung und Fehlerprüfung

C#

Eigenschaft, die nur Integerwerte zwischen 50 und 200 akzeptiert und gegebenenfalls eine entsprechende Anpassung vornimmt

```
        private int _MeineVar;
```

BEISPIEL 38.8: Wertebereichsbeschränkung und Fehlerprüfung

```
public int EigenschaftMitBereichsbeschränkung
{
    get { return _MeineVar; }
    set
    {
        if (value < 50)
            _MeineVar = 50;
        else
            if (value > 200)
                _MeineVar = 200;
            else
                _MeineVar = value;
    }
}
```

Alternativ können Sie auch den Anwender mit Fehlermeldungen zupflastern:

```
public int EigenschaftMitFehlerprüfung
{
    get { return _MeineVar; }
    set
    {
        if (value < 50)
            throw new ArgumentOutOfRangeException("Wert muss größer 50 sein!");
        else
            if (value > 200)
                throw new ArgumentOutOfRangeException("Wert muss kleiner 200 sein!");
            else
                _MeineVar = value;
    }
}
```

Ergebnis

Fehlermeldung bei Angabe eines falschen Wertes:

Eine weitere Möglichkeit zur Einschränkung bieten die so genannten Aufzählungstypen (Enumerationen), die im folgenden Abschnitt besprochen werden.

38.5.11 Eigenschaften von Aufzählungstypen

Ist der Wertebereich einer Eigenschaft zur Entwicklungszeit bereits bekannt, können Sie die Fehlermöglichkeiten durch Verwendung eines Aufzählungstyps einschränken.

Im Eigenschaftenfenster wird in diesem Fall eine Liste der zulässigen Werte angezeigt:

Der Vorteil für den Endanwender liegt auf der Hand: Statt irgendwelcher numerischer Werte erscheinen aussagefähige Beschriftungen. Auf die Verwendung der Hilfefunktion kann deshalb fast immer verzichtet werden.

Für Sie als Entwickler bedeutet ein Aufzählungstyp ein wenig mehr Arbeit, da Sie zuerst einen entsprechenden Typ deklarieren müssen. Die eigentliche Umsetzung in der Komponentendefinition unterscheidet sich nicht von der einer einfachen Eigenschaft.

BEISPIEL 38.9: Aufzählungseigenschaft deklarieren

C#

Typ deklarieren:

```
public enum MyEnum
 {
     Montag = 0,
     Dienstag = 1,
     Mittwoch = 2,
     Donnerstag = 3,
     Freitag = 4,
     Samstag = 5,
     Sonntag = 6,
 }
```

Private Variable:

```
private MyEnum myEnum;
```

Die Eigenschaft:

```
public MyEnum Aufzählungseigenschaft
```

BEISPIEL 38.9: Aufzählungseigenschaft deklarieren

```
        {
            get { return myEnum; }
            set { myEnum = value; }
        }
```

BEISPIEL 38.10: Verwendung der Eigenschaft *Aufzählungseigenschaft*

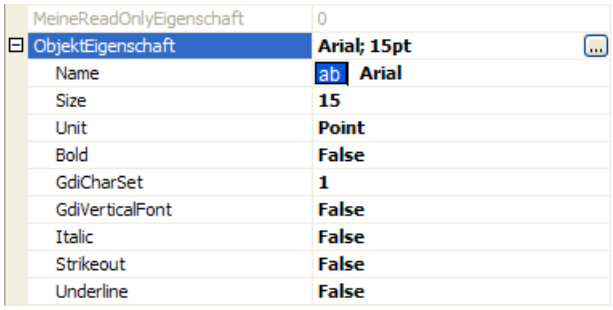
C#

```
...
eigenschaftenBeispiel1.Aufzählungseigenschaft =
                            Beispiel1.EigenschaftenBeispiel.MyEnum.Donnerstag;
```

38.5.12 Standard Objekt-Eigenschaften

Während Sie beim vorhergehenden Eigenschaftstyp lediglich einzelne Optionen festlegen können, bietet eine Objekt-Eigenschaft wesentlich mehr. Das wohl prominenteste Beispiel dürfte die Eigenschaft *Font* sein, die wiederum über eigene Eigenschaften verfügt.

Im Eigenschaftenfenster können Sie entweder einen Eigenschaften-Editor verwenden, oder Sie expandieren den Eintrag und legen die Objekt-Eigenschaften einzeln fest:



BEISPIEL 38.11: Implementieren eines *Font*-Objekts in unserer Beispielkomponente

C#

```
        private Font myFont = new Font("Arial",15);
        public Font ObjektEigenschaft
        {
            get { return myFont;  }
            set { myFont = value; }
        }
```

38.5.13 Eigene Objekt-Eigenschaften

Komplizierter wird die ganze Geschichte, wenn Sie ein neues Objekt erstellen wollen. In diesem Fall genügt es nicht, wenn Sie einfach ein Objekt in Ihre Komponente integrieren, wie es das folgende Beispiel zeigt.

BEISPIEL 38.12: Unsere Komponente hat ein Objekt mit drei Eigenschaften vom Typ *Integer*

C#

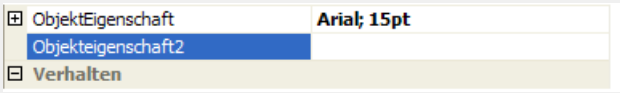
```
public class TestKlasse : ExpandableObjectConverter
{
    private int _Wert1;

    public int Wert1
    {
        get { return _Wert1; }
        set { _Wert1 = value; }
    }
    private int _Wert2;

    public int Wert2
    {
        get { return _Wert2; }
        set { _Wert2 = value; }
    }
    private int _Wert3;

    public int Wert3
    {
        get { return _Wert3; }
        set { _Wert3 = value; }
    }
}
```

Die Eigenschaft:

```
...
    private TestKlasse _myTestklasse;

    public TestKlasse Objekteigenschaft2
    {
        get { return _myTestklasse; }
        set { _myTestklasse = value; }
    }
```

Ergebnis

Ein Test der Komponente in der IDE zeigt folgendes Ergebnis:



Das Resultat dürfte sicher nicht ganz Ihren Erwartungen entsprechen, haben wir doch einen Fehler in unserem Beispiel, wie er gern gemacht wird:

HINWEIS: Die private Objektvariable ist nicht initialisiert.

BEISPIEL 38.12: Unsere Komponente hat ein Objekt mit drei Eigenschaften vom Typ *Integer*

Deshalb:

```
private TestKlasse _myTestklasse = new TestKlasse();
```

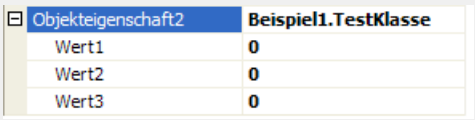
Ein erneuter Blick in das Eigenschaftenfenster zeigt keine Veränderung, aber die Komponente ist zumindest zur Laufzeit schon voll funktionstüchtig:

```
eigenschaftenBeispiel1.Objekteigenschaft2.Wert2 = 5;
this.Text = eigenschaftenBeispiel1.Objekteigenschaft2.Wert2.ToString();
```

Nach viel Sucherei in der Microsoft Dokumentation und in diversen Foren stellt sich heraus, dass wir nur mit zusätzlichen Attributen, die die Anzeige im Eigenschaftenfenster steuern, weiterkommen:

```
[TypeConverterAttribute(typeof(System.ComponentModel.ExpandableObjectConverter))]
public class TestKlasse
{
```

Das Ergebnis im Eigenschaftenfenster:



HINWEIS: Wem die Anzeige von "Beispiel1.TestKlasse" nicht gefällt, der kann sich einen eigenen *TypeConverter* von *ExpandableObjectConverter* ableiten und die Methoden *CanConvertFrom*, *ConvertFrom*, *ConvertTo*, *GetCreateInstanceSupported* sowie *Create Instance* überschreiben.

Damit dürften wir die wichtigsten Varianten von Eigenschaften berücksichtigt haben. Auf alle Möglichkeiten (Eigenschafteneditor etc.) können wir aus Platzgründen leider nicht eingehen, die gezeigten Beispiele dürften jedoch manche Unklarheit beseitigt haben.

38.6 Methoden

Hinter den Methoden von Steuerelementen verbirgt sich nichts anderes als normale Funktionen, die jedoch fest an die jeweilige Klasse gekoppelt sind. Aus der Realisierung ergibt sich auch das Einsatzgebiet: Methoden sollten, im Gegensatz zu Eigenschaften, Aktionen auslösen, die teilweise mit Rückgabewerten (Funktionen) verbunden sind. Methoden bieten sich auch dann an, wenn es darum geht, mehrere Eigenschaften gleichzeitig zu beeinflussen.

HINWEIS: Da Definition und Verwendung von Methoden zum Handwerkszeug des C#-Programmierers gehört, möchten wir im Weiteren nur noch auf einige spezielle Themen eingehen.

38.6.1 Konstruktor

Das Besondere: Diese Methode wird automatisch beim Erzeugen einer Klasseninstanz ausgeführt. Damit ist dies auch der ideale Ansatzpunkt, um

- alle internen Variablen unseres Steuerelements zu initialisieren,
- das Aussehen des Steuerelements anzupassen
- und gegebenenfalls Ereignishandler zuzuweisen.

HINWEIS: Der Konstruktor trägt immer den Namen der Klasse.

Eine Besonderheit gilt es noch zu beachten: Erzeugen Sie eine neue Komponentenklasse, legt C# automatisch zwei überladene Konstruktoren an, wie der folgenden Quellcodeausschnitt zeigt:

```
namespace Beispiel2
{
    public partial class Component1 : Component
    {
        public Component1()
        {
            InitializeComponent();
        }

        public Component1(IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }
    }
}
```

Welcher Konstruktor wird nun eigentlich ausgeführt?

Die Antwort: Wenn die Komponente einem Container (*Form*/*Panel* etc.) zugeordnet wird, nutzt die IDE nicht den einfachen, sondern den Konstruktor mit Parameter, um die Komponente in die *IContainer*-Auflistung einzufügen.

```
    partial class Form1
    {
        private System.ComponentModel.IContainer components = null;
...
        private void InitializeComponent()
        {
            this.component11 = new Beispiel2.Component1(this.components);
```

Andernfalls wird der Standard-Konstruktor verwendet:

```
    partial class Form1
    {
```

```
        private void InitializeComponent()
        {
...
            this.component11 = new Beispiel2.Component1();
```

Der Vorteil: Wird ein Formular erzeugt und klinkt sich die Komponente in die *IContainer*-Auflistung *components* ein, wird auch beim Aufruf von *Form.Dispose* die *Dispose*-Methode des Controls aufgerufen. Dazu überschreibt das Formular seine geerbte *Dispose*-Methode:

```
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
                components.Dispose();
            base.Dispose(disposing);
        }
```

Ressourcen werden so definiert freigegeben, andernfalls müsste nach dem Löschen des Formulars Ihre Komponente irgendwann vom Garbage-Collector entsorgt werden.

BEISPIEL 38.13: Formular öffnen und wieder freigeben

C#
```
        Form2 f2 = new Form2();
        f2.ShowDialog();
        f2.Dispose();            // --> Hier wird auch Component.Dispose() aufgerufen
```

38.6.2 Class-Konstruktor

Noch ein Konstruktor? Ja! Wer schon einmal mit dem NET-Reflector in einer NET-Assembly herumgestöbert hat, wir sicher auch die Methode *.cctor* gefunden haben. Hierbei handelt es sich um den **Class-Konstruktor**, der beim ersten Zugriff auf die Klasse ausgeführt wird. Der Einwand, das tut der normale Konstruktor auch, kann so nicht stehen bleiben. Was passiert beispielsweise, wenn Sie eine statische Methode aufrufen? In diesem Fall wird vorher automatisch der Class-Konstruktor abgearbeitet (nach dem Laden der Klasse, vor dem Zugriff auf die Member), der eigentliche Konstruktor ist zu diesem Zeitpunkt noch gar nicht in Aktion getreten.

Womit auch gleich das Anwendungsgebiet ersichtlich wird. Nutzen Sie diesen Konstruktor, um statische Eigenschaften zu initialisieren.

BEISPIEL 38.14: Initialisieren der Eigenschaft *Startzeit*

C#
```
    public partial class Component1 : Component
    {
        static public DateTime Startzeit;
        static Component1()                          // Class-Konstruktor
        {
            Startzeit = System.DateTime.Now;
        }
...
```

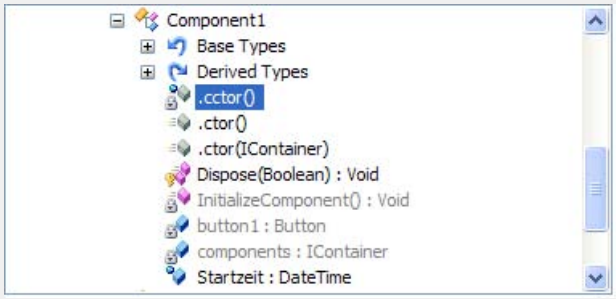
HINWEIS: Beachten Sie, dass die Eigenschaft *Startzeit* für alle späteren Instanzen der Klasse den gleichen Wert hat (es handelt sich eben um eine Klasseneigenschaft).

BEISPIEL 38.15: Die Verwendung im aufrufenden Programm

C#

```
private void Form1_Load(object sender, EventArgs e)
{
    Text = Component1.Startzeit.ToString();
}
```

Ergebnis



38.6.3 Destruktor

Da sich in .NET bekanntlich der Garbage Collector um die endgültige Zerstörung von nicht mehr benötigten Objekten kümmert, wird ein Destruktor im herkömmlichen Sinn nicht mehr gebraucht. Möchten Sie dennoch auf des relativ unbestimmte Ende Ihres Steuerelements reagieren, können Sie eine private Methode mit dem Namen der Klasse erstellen. Zur Unterscheidung vom Konstruktor wird ein "~"-Zeichen vorangestellt.

BEISPIEL 38.16: Destruktor

C#

```
~MyEdit()
 {
   Debug.WriteLine(" Ich bin am Ende");
 }
```

HINWEIS: Den Destruktor selbst können Sie nicht per Code aufrufen!

38.6.4 Aufruf des Basisklassen-Konstruktors

HINWEIS: Wollen/müssen Sie einen bestimmten Konstruktor der Basisklasse aufrufen, verwenden Sie *:base(Parmeterliste)*.

BEISPIEL 38.17: Verwendung von *base*

C#
```
public MyEdit(int i) : base(i)
{
   ...
}
```

Statt des Basisklassen-Standard-Konstruktors wird im obigen Beispiel der Konstruktor mit dem Integer-Parameter verwendet.

38.6.5 Aufruf von Basisklassen-Methoden

Müssen Sie eine Methode der Basisklasse aufrufen, verwenden Sie den Bezeichner *base*.

BEISPIEL 38.18: Überschreiben von *OnPaint* und Aufruf der Basisklassen-Methode

C#
```
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);
    e.Graphics.DrawLine(Pens.Red, 10, 10, 100, 100);
    ...
}
```

38.7 Ereignisse (Events)

Nicht ganz so einfach wie das Programmieren von Methoden ist die Realisierung von Ereignissen bzw. Ereignisprozeduren. Events stellen einen Mechanismus dar, der ein externes Ereignis (zum Beispiel eine Windows-Botschaft) oder ein Nutzerereignis (zum Beispiel der Klick auf einen Button) mit einer eigenen Routine (Eventhandler) verbindet.

HINWEIS: Eine ausführliche Einführung in das Ereignismodell von Microsoft finden Sie im Kapitel 26.

Fünf Schritte sind für das Implementieren eines Events erforderlich:

- eventuell eine neue Klasse für den *EventArgs*-Parameter (diese wird von *System.Event-Args* abgeleitet) definieren,
- eine Delegate-Deklaration (Definition des Ereignistyps),
- ein interner Delegate für den Event,
- eine Event-Deklaration,
- das eigentliche Auslösen des Ereignisses.

38.7.1 Ereignis mit Standardargument definieren

Relativ einfach ist das Implementieren eines Ereignisses, das sich mit den Standardargumenten vom Typ *System.EventArgs* zufrieden gibt. Allerdings sollten Sie sich schon hier mit einigen Grundkonventionen und Eigenheiten vertraut machen.

BEISPIEL 38.19: Ereignis mit Standardargument definieren

C#

Anhand einer *TextBox*, die ein zusätzliches Ereignis erhält, sollen die wichtigsten Schritte erläutert werden.

```
public partial class MyEdit : TextBox
{
    public MyEdit()
    {  InitializeComponent(); }
```

Zunächst instanziieren wir das Ereignis *TextboxFull* auf Basis des *EventHandler*-Delegaten:

```
    public event EventHandler TextboxFull;
```

Die Deklaration einer Ereignismethode[1], in welcher das Ereignis ausgelöst wird:

```
    protected void OnTextBoxFull()
    {
```

Nur wenn dem Ereignis im Programm auch eine Ereignisprozedur zugewiesen wurde, wird auch das Ereignis ausgelöst:

```
        if (TextboxFull != null)
        {
            EventArgs args = new EventArgs();
            TextboxFull(this, args);          // Ereignis wird ausgelöst
        }
    }
```

HINWEIS: Microsoft empfiehlt, beim Ableiten von Komponenten statt der Verwendung eines Eventhandlers besser die Ereignismethode zu überschreiben.

Hier überwachen wir die *TextBox* und rufen gegebenenfalls die Methode *OnTextBoxFull* auf:

```
    protected override void OnTextChanged(EventArgs e)
    {
        if (this.Text.Length > 8) OnTextBoxFull();
        base.OnTextChanged(e);
    }
```

HINWEIS: Vergessen Sie nicht *base.OnTextChanged* aufzurufen, anderenfalls fällt das Ereignis *TextChanged* unter den Tisch!

[1] Entsprechend der Microsoft-Konvention sollten die Ereignismethoden immer mit *On...* beginnen.

BEISPIEL 38.19: Ereignis mit Standardargument definieren

Im Anwenderprogramm findet sich jetzt eine neues Ereignis:

```
private void myEdit1_TextboxFull(object sender, EventArgs e)
{
    MessageBox.Show("Hilfe nicht soviel Buchstaben ...");
}
```

38.7.2 Ereignis mit eigenen Argumenten

Etwas aufwändiger ist die Definition eigener Argumenttypen.

BEISPIEL 38.20: Der Parameter soll zusätzlich eine Message an das aufrufende Programm zurückgeben.

C#

Leiten Sie dazu eine Klasse vom Typ *System.EventArgs* ab:

```
class MyEventArgs : EventArgs
{
    private String _Message;
```

Die neue Eigenschaft für das Argument:

```
    public String Message
    {
        get { return _Message; }
        set { _Message = value; }
    }
```

Implementieren Sie gegebenenfalls einen neuen Konstruktor für das neue Argument:

```
    public MyEventArgs(String msg)
    {  _Message = msg;  }
}
```

Die eigentliche Komponente:

```
public partial class MyEditMsg : TextBox
{
```

Fügen Sie der Komponente einen Delegaten für den obigen Ereignis-Typ hinzu:

```
    public delegate void TextBoxFullHandler(object sender, MyEventArgs e);
```

Die Deklaration des Ereignisses:

```
    public event TextBoxFullHandler TextBoxFull;
```

Alternativ können Sie auch die folgende Deklaration nutzen:

```
    public event EventHandler<MyEventArgs> TextBoxFull;
```

Die weitere Verwendung entspricht der bisherigen Vorgehensweise, beim Aufruf des Delegaten wird jetzt jedoch ein anderer Konstruktor genutzt:

BEISPIEL 38.20: Der Parameter soll zusätzlich eine Message an das aufrufende Programm zurückgeben.

```
        protected void OnTextBoxFull()
        {
            if (TextBoxFull != null)
            {
                MyEventArgs args = new MyEventArgs("Ich bin voll!!!!!!");
                TextBoxFull(this, args);
            }
        }
        protected override void OnTextChanged(EventArgs e)
        {
            if (this.Text.Length > 8) OnTextBoxFull();
            base.OnTextChanged(e);
        }
    }
```

Das war es auch schon, im Programm können Sie den Parameter wie folgt verwenden:

```
        private void myEditMsg1_TextBoxFull(object sender, MyEventArgs e)
        {
            MessageBox.Show(e.Message);
        }
```

38.7.3 Ein Default-Ereignis festlegen

Ausnahmsweise ist diese Aufgabe mit einer einzigen Zeile "Code" erledigt. Fügen Sie einfach vor die betreffende Klassendefinition das Attribut *[DefaultEvent]* ein.

BEISPIEL 38.21: Default-Ereignis definieren

C#

```
    [DefaultEvent("TextBoxFull")]
    public partial class MyEditMsg : TextBox
    {
        public delegate void TextBoxFullHandler(object sender, MyEventArgs e);
        public event TextBoxFullHandler TextBoxFull;
```

Nach einem Doppelklick auf die spätere Komponente wird jetzt automatisch eine Ereignismethode für *TextBoxFull* erzeugt.

38.7.4 Mit Ereignissen auf Windows-Messages reagieren

Ganz zum Schluss wollen wir Sie mit einem nicht minder interessanten Thema foltern. Es geht um die guten alten Windows-Ereignisse bzw. Botschaften. Auch wenn das .NET-Framework für fast alle Eventualitäten ein(e) Methode/Ereignis im Stammbaum seiner Forms/Controls vorhält, es gibt Fälle, wo wir direkt auf derartige Ereignisse reagieren wollen.

Anlaufpunkt für den .NET-Programmierer ist die Methode *WndProc*. Da stutzt sicher jeder altgediente Win32-Programmierer. Diese Variante, in die Windows-Botschaftsbehandlung einzugreifen, dürfte vielen bekannt vorkommen. Schon damals wurde die Methode einfach überschrieben, um neue Funktionalität hinzuzufügen. Ein kleines Beispiel soll Ihnen den Übergang in die .NET-Welt demonstrieren.

BEISPIEL 38.22: Auf Windows-Messages reagieren

C#

Ein *TextBox*-Nachfahre soll auf das Mausrad reagieren[1]. Ein eingetragener Wert ist jeweils zu inkrementieren bzw. zu dekrementieren. Zwei neue Ereignisse sollen uns über den Mausrad-Status informieren.

Unsere Komponente:

```
public partial class WheelEdit : TextBox
{
```

Die beiden Ereignisse deklarieren:

```
    public event EventHandler WheelUp;
    public event EventHandler WheelDown;
```

Und ab geht es in die gute alte Win32-Welt mit ihren Konstanten und verschachtelten Parametern:

```
    private const int WM_MOUSEWHEEL = 522;
    public WheelEdit()
    {
        InitializeComponent();
    }
```

Überschreiben der Botschaftsbehandlung:

```
    protected override void WndProc(ref Message m)
    {
```

Die eigentliche Message herausfiltern:

```
        switch (m.Msg)
        {
            case WM_MOUSEWHEEL:
```

Den Inhalt der *TextBox* ermitteln:

```
                int v = Convert.ToInt32(this.Text);
```

Über den Parameter *m.WParam* können wir die Drehrichtung bestimmen:

```
                if ((((Int32)m.WParam) >> 31) == 0)
                {
                    this.Text = (v + 1).ToString();
```

Auslösen der Ereignisse:

[1] Natürlich geht es auch anders, aber so haben wir ein halbwegs sinnvolles Beispiel für die Messageverarbeitung.

BEISPIEL 38.22: Auf Windows-Messages reagieren

```
                    if (WheelUp != null) WheelUp(this, new EventArgs());
                }
                else
                {
                    this.Text = (v - 1).ToString();
                    if (WheelDown != null) WheelDown(this, new EventArgs());
                }
                this.Invalidate();
                break;
            default:
                break;
        }
```

Das sollten Sie in keinem Fall vergessen:

```
            base.WndProc(ref m);
        }
...
```

HINWEIS: Mit obigem Codegerüst können Sie auch andere Botschaften, die von der Komponente empfangen werden, auswerten.

Damit genug zu den diversen Komponententypen, auch wenn es noch reichlich zu diesem Thema zu sagen gäbe (von datengebundenen Komponenten ganz zu schweigen). Wenden wir uns nun noch einigen spezielleren Themen zu ...

38.8 Weitere Themen

Dieser Abschnitt fast in loser Folge einige interessante Themen rund um die Komponentenprogrammierung zusammen.

38.8.1 Wohin mit der Komponente?

In den bisherigen Beispielen haben wir es uns einfach gemacht und unsere Komponenten jeweils in das Windows-Projekt mit aufgenommen. Das ist zwar sehr praktisch, wenn Sie Komponenten testen wollen, aber im Normalfall soll die Komponente ja in mehreren Anwendungen eingesetzt werden.

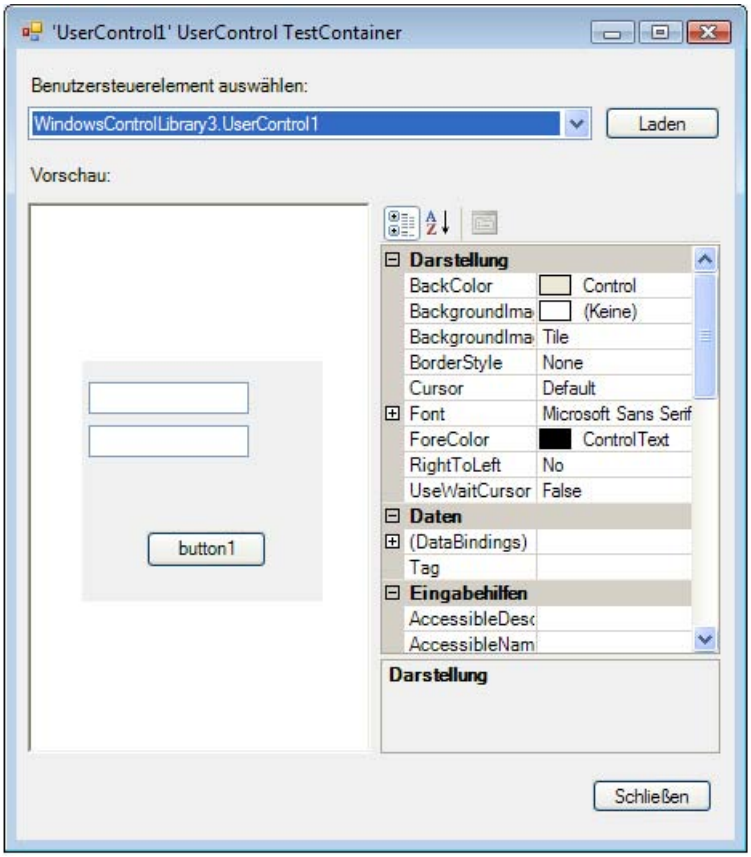
Zwei Varianten bieten sich beim Blick in den Dialog "Neues Projekt"an:

- Klassenbibliothek
- Windows Forms-Steuerelementebibliothek

Worin liegt der Unterschied?

Zunächst die allgemeine Antwort: Sie können beide Projektarten für Ihre Steuerelemente verwenden. In jedem Fall wird eine Assembly mit *.dll*-Extension erzeugt.

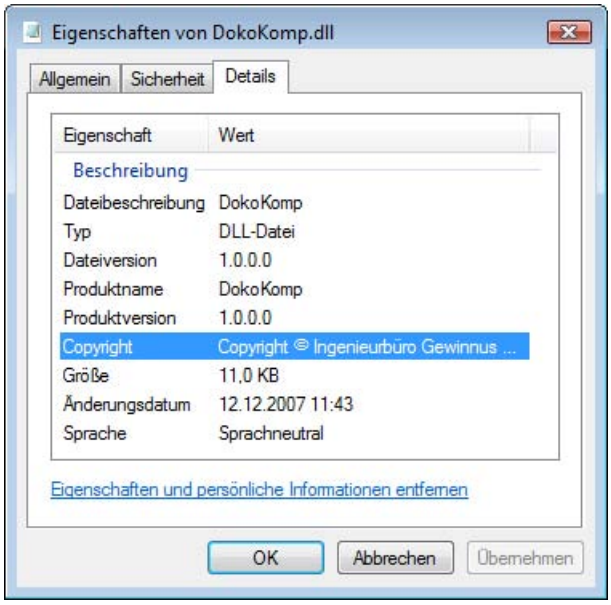
Die Unterschiede zeigen sich bei den Feinheiten. So bietet die "Windows Forms-Steuerelementebibliothek" bereits eingebettete Ressourcen sowie einen einfachen Test-Container für Ihre User-Controls. Diesen starten Sie wie gewohnt mit F5, ein weiterer Unterschied zu einer einfachen Klassenbibliothek, die Sie zwar erstellen, aber nicht "ausführen" können.



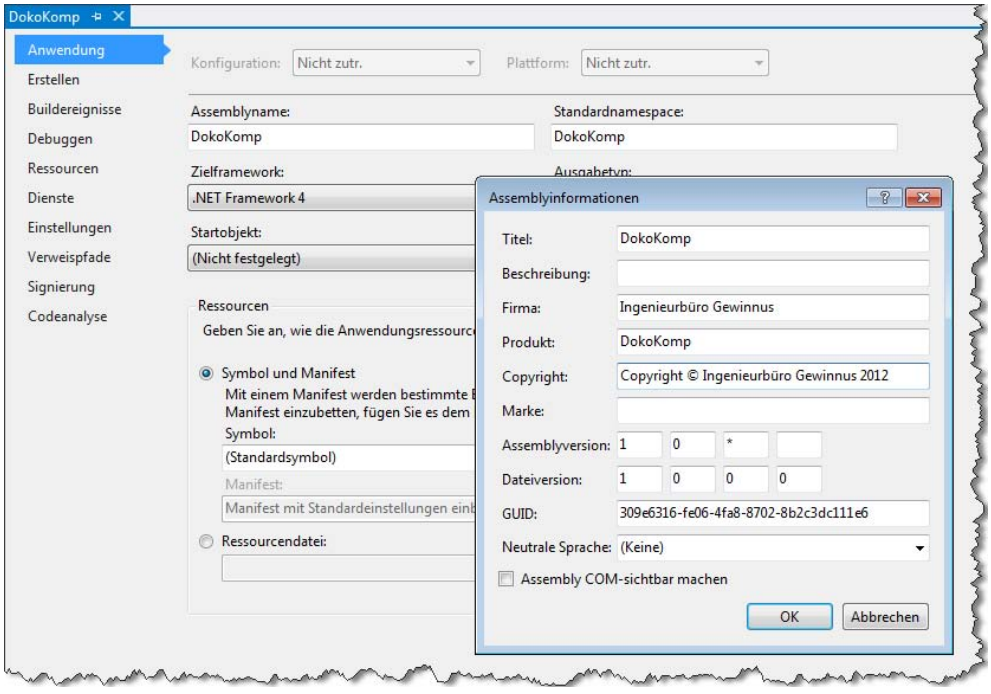
Wie Sie obiger Abbildung entnehmen, beschränkt sich der Test-Container auf die Anzeige der verfügbaren Eigenschaften. Zum Testen Ihrer Komponente ist das sicher ein nettes Feature.

38.8.2 Assembly-Informationen festlegen

In einer Assembly werden neben dem reinen Programmcode auch weitere Informationen abgelegt, die unter anderem für das Erzeugen von *Strong Names* (Starke Namen) Verwendung finden. Doch auch der normale Anwender Ihrer Assembly kann von diesen Informationen profitieren, indem er sich über das Kontextmenü (im Windows Explorer) die Eigenschaften anzeigen lässt.



Die entsprechenden Informationen können Sie in Visual Studio über einen Projekteigenschaften-Dialog festlegen:



Alternativ können Sie Assembly-Infos auch direkt in die Datei *AssemblyInfo.cs* eintragen.

BEISPIEL 38.23: ***AssemblyInfo.cs***

C#
```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// Allgemeine Informationen über eine Assembly werden über die folgenden
// Attribute gesteuert. Ändern Sie diese Attributwerte, um die Informationen zu ändern,
// die mit einer Assembly verknüpft sind.

[assembly: AssemblyTitle("ClassLibrary1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Ingenieurbüro Gewinnus")]
[assembly: AssemblyProduct("ClassLibrary1")]
[assembly: AssemblyCopyright("Copyright © Ingenieurbüro Gewinnus 2006")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Durch Festlegen von ComVisible auf "false" werden die Typen in dieser Assembly unsichtbar
// für COM-Komponenten. Wenn Sie auf einen Typ in dieser Assembly von
// COM zugreifen müssen, legen Sie das ComVisible-Attribut für diesen Typ auf "true" fest.
[assembly: ComVisible(false)]

// Die folgende GUID bestimmt die ID der Typbibliothek, wenn dieses Projekt für COM verfügbar
gemacht wird
[assembly: Guid("8093140f-abea-4e27-84a9-4763dfb2844b")]

// Versionsinformationen für eine Assembly bestehen aus den folgenden vier Werten:
//
//      Hauptversion
//      Nebenversion
//      Buildnummer
//      Revision
//
// Sie können alle Werte angeben oder die standardmäßigen Revisions- und Buildnummern
// übernehmen, indem Sie "*" eingeben:

[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

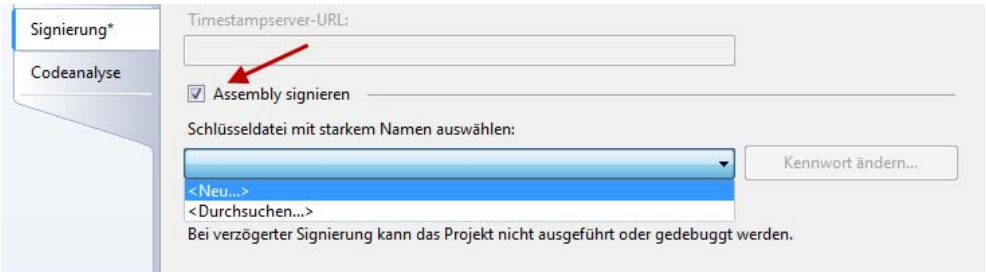
In diesem Zusammenhang ist auch die Bezeichnung *Strong Name* von Bedeutung. Dieser setzt sich aus dem Namen, der Versionsnummer, der Kulturinformation sowie einem öffentlichen Schlüssel und einer digitalen Signatur zusammen. Wie Sie diesen Schlüssel erzeugen, um einen *Strong Name* zu generieren, zeigt der folgende Abschnitt.

38.8.3 Assemblies signieren

Sicher stellt sich auch Ihnen die Frage, ob Sie Ihre Assembly signieren müssen oder nicht. Eine kurze Antwort darauf: Ihre Assembly **müssen** Sie in jedem Fall signieren, wenn Sie diese im *Global Assembly Cache* (GAC) ablegen wollen. Der erzeugte *Strong Name* stellt einen eindeutigen Bezeichner für Ihre Assembly dar.

Erstellen eines AssemblyKey-File

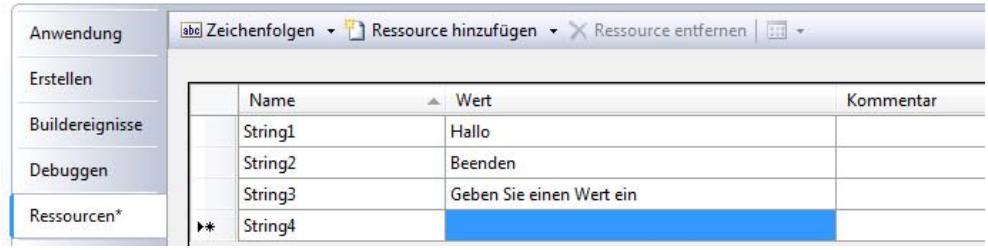
Die zur Signierung erforderliche Schlüsseldatei mit dem Schlüsselpaar wird mit dem NET-Kommandozeilentool *sn.exe* erzeugt. Seit Visual Studio 2005 ist die entsprechende Funktionalität auch über die Projekt-Eigenschaften verfügbar:



Hier erfolgt auch gleich die Verknüpfung mit dem Projekt, Sie brauchen sich also nicht mehr mit Quellcode oder der Kommandozeile herumzuplagen.

38.8.4 Komponenten-Ressourcen einbetten

Eine Komponente besteht meist nicht nur aus Quellcode, sondern erfordert auch die Ausgabe von Grafiken und Sound bzw. diversen Zeichenfolgen etc. Alle diese "Ressourcen" können ganz normal in eine zum Projekt gehörende Ressourcendatei (*Resources.resx*) eingebettet werden und stehen über die Projekteigenschaften zur Verfügung:



Der Zugriff in der Komponente erfolgt dann wie gewohnt per *Properties.Resources*....

38.8.5 Der Komponente ein Icon zuordnen

Nachdem wir uns nun schon geraume Zeit mit der Komponenten-Programmierung herumgeschlagen haben, kommt bei den Ästheten unter den Lesern sicher bald auch der Wunsch nach einer besseren Optik für die selbst erstellten Komponenten auf. Das freudlose Icon, dass standardmäßig eingeblendet wird, ist wenig informativ und wohl auch nicht jedermanns Sache.

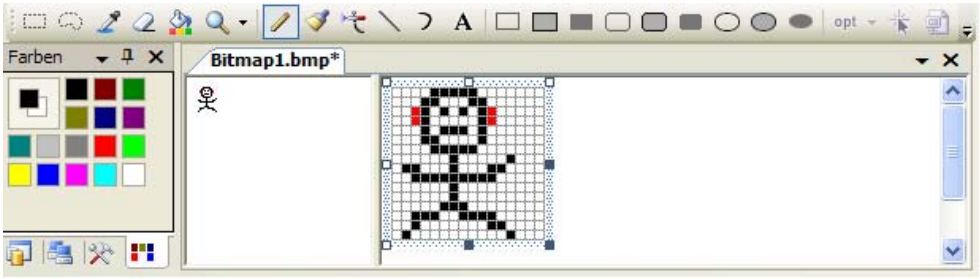
Icon erstellen

Erzeugen (oder kopieren) Sie zunächst eine 16x16 Pixel große Bitmap-Datei. Dazu können Sie zum Beispiel die Visual Studio-Bordmittel verwenden (*Einfügen|Neues Element Bitmap*).

Speichern Sie jetzt die Datei im Projekt ab (als eingebettete Ressource). Doch schon an dieser Stelle sind einige Konventionen einzuhalten. So leitet sich der Dateiname direkt von der betreffenden Komponentenklasse ab:

```
<Klassenname>.bmp
```

Für unser kleines Beispiel lautet der Name also *WheelEdit.bmp*.



HINWEIS: Alternativ können Sie auch das *ToolboxBitmap*-Attribut für die Komponentenklasse verwenden. In diesem Fall brauchen Sie sich nicht auf den Klassennamen zu beschränken.

Nach dem Kompilieren und Einbinden in die Visual Studio-IDE sollte sich Ihnen etwa der folgende Anblick bieten:

38.8.6 Den Designmodus erkennen

Nicht immer läuft Ihre Komponente nur im späteren Programm des Anwenders. Auch im Visual Studio-Form-Designer sollte die Komponente "eine gute Figur machen". Meist stehen zu diesem Zeitpunkt nicht alle Ressourcen zur Verfügung oder das Verhalten der Komponente soll an den Designer angepasst werden. Doch wie können Sie als Entwickler den Designmodus eigentlich vom normalen Laufzeitmodus unterscheiden?

Die Antwort ist recht einfach, die Eigenschaft *DesignMode* liefert uns die gewünschte Information.

BEISPIEL 38.24: Unterscheidung Entwurfsmodus/Laufzeitmodus

C#

```
if (this.DesignMode)
{
    g.DrawImage(bmp, 0, 0);
}
else
{
    g.DrawImage(bmp, 0, 0);
    ImageAnimator.UpdateFrames();
}
```

38.8.7 Komponenten lizenzieren

Da mühen Sie sich mit dem Programmieren von Komponenten ab und wollen vielleicht sogar ein paar davon verkaufen. Zwangsläufig kommt dann die Frage auf, wie Sie sich davor schützen können, dass die Komponenten einfach weitergegeben werden, ohne dass der zukünftige Nutzer (Entwickler) dafür etwas bezahlt. Dass eine Firma wie Microsoft auch an diese Thematik gedacht hat, dürfte auf der Hand liegen. Das .NET-Framework bringt bereits einige rudimentäre Funktionen für die Lizenzierung von Komponenten mit. Allerdings dürfte dies in den meisten Anwendungsfällen wohl eher als erste Anregung denn als Lösung verstanden werden, haben wir es doch hier mit Entwicklern zu tun, die auch mal einen Blick in den Code werfen (können).

Vier Varianten bieten sich an:

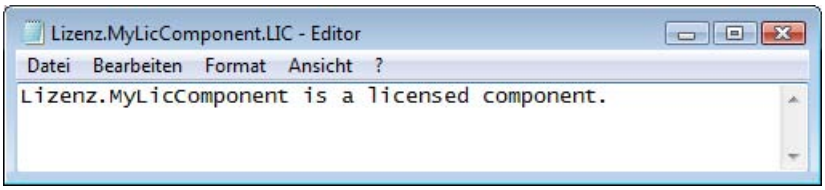
- Sie verwenden den *LicFileLicenseProvider* und eine Lizenzdatei um minimale Lizenzierungsfunktionen zu implementieren.
- Sie leiten den *LicFileLicenseProvider* ab und implementieren einen eigenen Provider mit eigenen Prüfalgorithmen (Lizenzdatei).
- Sie erstellen sowohl einen eigenen License-Provider, als auch eine eigene *Licence*-Klasse (freie Wahl wo und wie der Key gespeichert wird).
- Sie vergessen die Microsoft-Klassen und kümmern sich um alles selbst.

Wir wollen im Rahmen dieses Abschnitts nur auf die ersten beiden Fälle eingehen und eine Anregung geben, wie Sie das System verbessern können.

Verwendung der LicFileLicenseProvider-Klasse

Wir bohren zunächst das Brett an der dünnsten Stelle und nutzen die von .NET bereitgestellten Möglichkeiten.

- Erstellen Sie eine neue Klassenbibliothek (z.B. mit dem Namen *Lizenz*) und Ihre Komponente (z.B. mit dem Namen *MyLicComponent*).
- Erzeugen Sie nachfolgend eine Textdatei (Notepad) mit folgendem Inhalt:

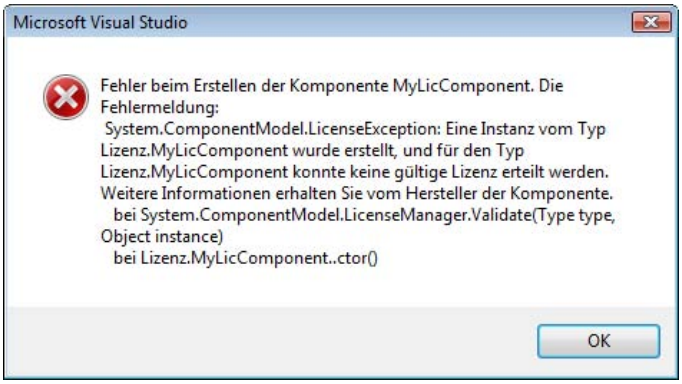


- Der Dateiname setzt sich aus dem Assemblynamen und dem Klassennamen der Komponente zusammen.
- Erweitern Sie Ihre Komponente um die im Folgenden fett hervorgehobenen Anweisungen:

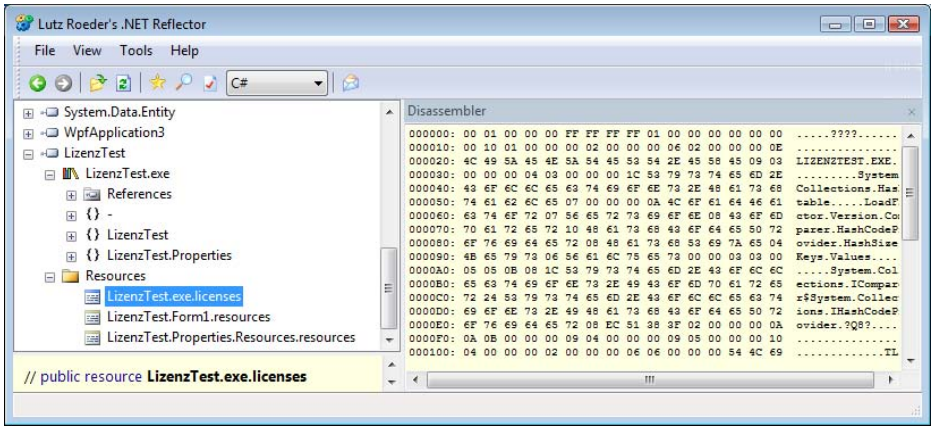
```
namespace Lizenz
{
// Provider zuweisen:
    [LicenseProvider(typeof(LicFileLicenseProvider))]
    public partial class MyLicComponent2 : Component
    {
// Licence-Instanz
        private License myLicence;

        public MyLicComponent2()
        {
            try
            {
                myLicence = LicenseManager.Validate(this.GetType(), this);
            }
            catch
            {
                MessageBox.Show("Fehlende Lizenz!");
                throw;
            }
            InitializeComponent();
        }
```

- Kompilieren Sie die Assembly und fügen Sie diese und die Lizenzdatei in das *bin*-Verzeichnis des finalen Projekts ein.

- Installieren Sie die Komponente in der Visual Studio-Toolbox und versuchen Sie die Komponente in ein Formular einzufügen. Solange die Lizenzdatei vorhanden ist, funktioniert dies auch, andernfalls erscheint unsere Fehlermeldung und dann der folgende Laufzeitfehler:



Geben Sie die kompilierte Anwendung weiter, ist keine Lizenzdatei mehr notwendig. Doch eine Unterscheidung zwischen Laufzeit- und Entwurfsmodus haben wir eigentlich gar nicht vorgenommen. Ein Blick in die Assembly zeigt, warum es mit der EXE läuft:



Ganz unbemerkt hat der Compiler die erforderlichen Lizenzen in einer Datei *licences.licx* gesammelt und als Ressource in die EXE kompiliert. Von dort werden sie dann auch geladen.

HINWEIS: Über die Sicherheit dieser Variante brauchen wir wahrscheinlich nicht zu streiten.

Ableiten der LicFileLicenseProvider-Klasse

Eine etwas sicherere Variante können Sie durch einfaches Ableiten der Klasse *LicFileLicenseProvider* erreichen. Überschreiben Sie einfach die Methode *IsKeyValid* und erstellen Sie Ihren eigenen Algorithmus:

```
public class MyPrivatLicenseProvider : LicFileLicenseProvider
{
    protected override bool IsKeyValid(string key, Type type)
    {
        if (key.Substring(2, 1) == "G")
            return true;
        else
            return false;
    }
}
```

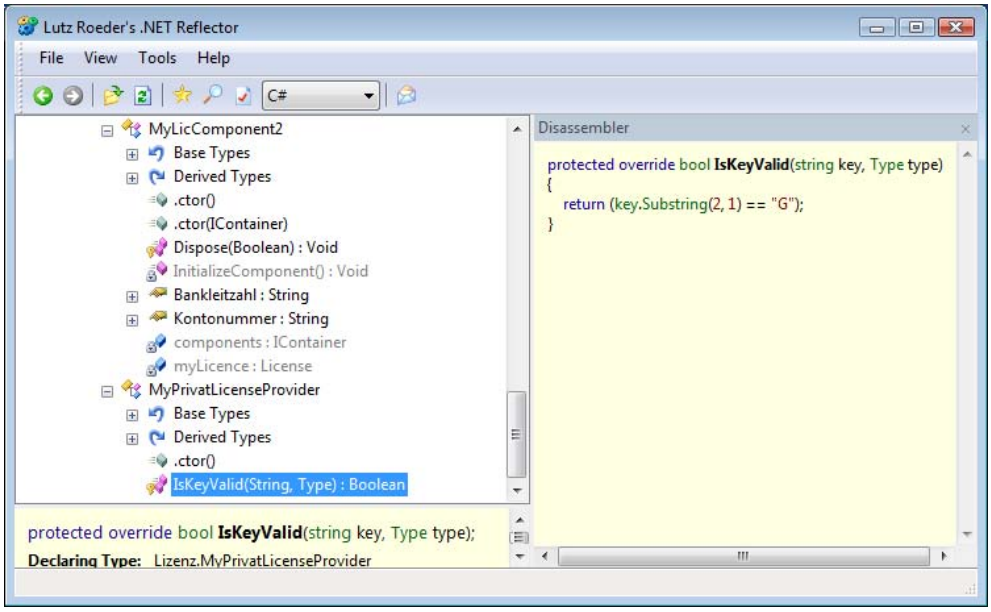
In diesem Fall muss an der dritten Stelle in der Textdatei ein großes "G" stehen.

Für Ihre Komponente ist von Ihnen nun noch der neue *LicenceProvider* vorzugeben:

```
namespace Lizenz
{
    [LicenseProvider(typeof(MyPrivatLicenseProvider))]
    public partial class MyLicComponent2 : Component
```

Das waren bereits die Änderungen zum Standardverfahren, der Rest läuft wie im vorhergehenden Abschnitt beschrieben.

HINWEIS: Bevor Sie jetzt viel Zeit für den Algorithmus verschwenden, nutzen Sie ruhig mal den .NET-Reflector, um in die Komponenten-Assembly zu schauen. Der angezeigte Quellcode dürfte Ihnen sicher bekannt vorkommen:



Bemerkung

Wer mit den beiden genannten Varianten nicht ganz zufrieden ist, der kann sich – wenn er etwas Arbeit investiert – auch selbst helfen:

- Verkaufen Sie zusammen mit der Komponente einen Key (GUID).
- Prüfen Sie beim Erstellen der Komponenten (Konstruktor) auf den Designmodus.
- Handelt es sich um diesen, lesen Sie die Mac-Adresse der Netzwerkkarte aus (oder eine andere eindeutige ID).
- Lassen Sie per Dialogfenster die GUID eingeben und übertragen Sie diese zusammen mit der Mac-Adresse an einen Web-Service.
- Im Webservice prüfen Sie zunächst, ob die GUID stimmt und speichern die Mac-Adresse ab.
- Senden Sie per Webservice einen eindeutigen Schlüssel (kann auch RSA verschlüsselt sein) an den Client zurück und speichern diesen in einer Lizenzdatei.
- Die Lizenzdatei ermöglicht dann die Prüfung der Lizenz, weitere Verbindungen zum Server sind nicht notwendig.
- Wenn jemand mehr als dreimal versucht, die Komponente auf verschiedenen Rechnern zu installieren, können Sie per GUID den Kunden ermitteln bzw. die Registrierung dauerhaft unterbinden.

HINWEIS: Vergessen Sie aber vor lauter Lizenzierung nicht die Programmierung der eigentlichen Komponenten!

38.9 Praxisbeispiele

38.9.1 AnimGif – Anzeige von Animationen

Mit dem folgenden einfachen Beispiel möchten wir Ihnen die bisherigen Ausführungen zur Komponentenentwicklung noch einmal im Zusammenhang demonstrieren.

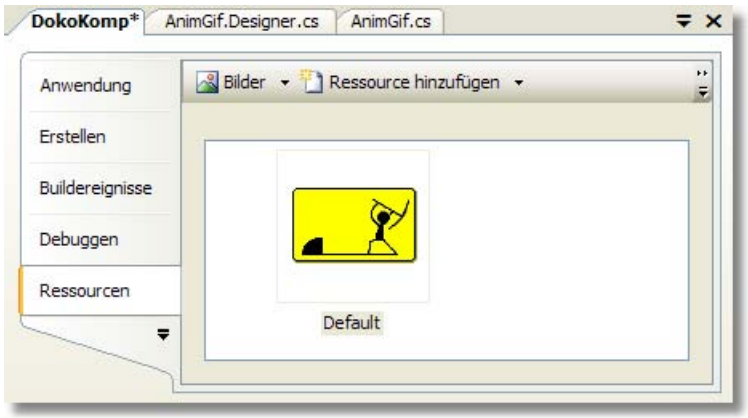
Ziel ist das Erstellen einer Komponente zur Anzeige animierter GIF-Grafiken. Neben dem standardmäßig mitgelieferten Bild soll der Anwender auch eigene Grafiken zuweisen können. Die Komponente soll sich daraufhin an die Abmessungen der Grafik anpassen. Auf diverse Extras, wie Ein-/Ausschalten etc., verzichten wir an dieser Stelle.

Oberfläche/Ressourcen

Erstellen Sie zunächst eine neue Klassenbibliothek und fügen Sie ein *Benutzerdefiniertes Steuerelement* hinzu. Speichern Sie dieses unter dem Namen *AnimGif* ab.

Nachfolgend können Sie sich im Internet auf die Suche nach einer Default-Grafik für Ihr neues Control machen. Unter den Stichworten "Animated GIF" werden Sie ganz sicher fündig, es gibt Tausende derartiger "Nervtöter".

Speichern Sie die Grafik auf Ihrem PC ab. Über den Menüpunkt *Projekt|Eigenschaften* können Sie die Grafik den Assembly-Ressourcen zuordnen. Vergeben Sie hier den Namen *Default*:



Damit sind die "Oberflächlichkeiten" abgeschlossen und wir können uns den Innereien zuwenden.

Quelltext

```
namespace DokoKomp
{
```

Da der Vorfahrtyp *Control* bereits alle Eigenschaften aufweist, die für uns wichtig sind, belassen wir es bei diesem Typ:

```
    public partial class AnimGif : Control
    {
```

Für die aktuelle Grafik eine private Variable:

```
        private Bitmap bmp;
```

Der Konstruktor kümmert sich um das Auslesen der Grafik aus den Ressourcen und die Vorbereitungen für eine flackerfreie Darstellung (*Double Buffering*):

```
        public AnimGif()
        {
            InitializeComponent();
            SetStyle(ControlStyles.UserPaint, true);
            SetStyle(ControlStyles.AllPaintingInWmPaint, true);
            SetStyle(ControlStyles.DoubleBuffer, true);
            bmp = Properties.Resources.Default;
        }
```

Die neue Eigenschaft *Gif* ermöglicht es dem Anwender, eine neue Grafik zuzuordnen:

```
public Bitmap Gif
{
    get { return bmp; }
    set
    {
        bmp = value;
        if (bmp == null) bmp = Properties.Resources.Default;
```

Größenanpassung an die Grafik:

```
        this.Width = bmp.Width;
        this.Height = bmp.Height;
    }
}
```

HINWEIS: Wird die Eigenschaft zurückgesetzt, d h. der Anwender weist keine Grafik zu, nehmen wir automatisch die Defaultgrafik aus den Ressourcen.

Mit dem Initialisieren des Controls wird es für uns ernst:

```
protected override void InitLayout()
{
    base.InitLayout();
```

Handelt es sich nicht um den Designmodus und ist die Grafik animierbar (was für ein Wort!), dann nutzen wir die Klasse *ImageAnimator*, um ein automatisches Umschalten der Einzelbilder per integrierten Timer zu erreichen:

```
    if ((!this.DesignMode)&(ImageAnimator.CanAnimate(bmp)))
        ImageAnimator.Animate(bmp, this.OnNextFrame);
}
```

Diese Methode wird immer aufgerufen, wenn ein neuer Frame angezeigt werden soll:

```
private void OnNextFrame(object o, EventArgs e)
{
    this.Invalidate();
}
```

Die Hauptarbeit leistet die überschriebene *OnPaint*-Methode:

```
protected override void OnPaint(PaintEventArgs pe)
{
    Graphics g = pe.Graphics;
```

Im Designmodus wird immer dasselbe Bild angezeigt:

```
    if (this.DesignMode)  g.DrawImage(bmp, 0, 0);
```

Andernfalls schalten wir zum nächsten Frame um:

```
    else
```

```
            {
                g.DrawImage(bmp, 0, 0);
                ImageAnimator.UpdateFrames();
            }
        }
    }
}
```

Test

Kompilieren Sie die Klassenbibliothek und fügen Sie die Komponente in die Toolbox ein. In einem neuen Windows Forms-Projekt können Sie sich von der Funktionstüchtigkeit überzeugen.

38.9.2 Eine FontComboBox entwickeln

Zum Schluss noch ein "quick and dirty"-Beispiel.

Lassen Sie uns die *ComboBox* so modifizieren, dass später ohne zusätzlichen Programmieraufwand alle verfügbaren Schriftarten angezeigt und als fertiger Font abgerufen werden können.

Quelltext

Erster Schritt zur fertigen Komponente ist das Ableiten von einer bereits vorhandenen. Öffnen Sie dazu ein neues Projekt und fügen Sie über den Menüpunkt *Projekt|Neues Element hinzufügen|Benutzerdefiniertes Steuerelement* ein neues Steuerelement in das Projekt ein.

Das bereits bestehende Code-Gerüst ändern Sie dahingehend, dass Sie statt der Basisklasse *Control* eine *ComboBox* verwenden:

```
public partial class FontCombo : ComboBox
```

Die Klasse im Überblick:

```
    public partial class FontCombo : ComboBox
    {
```

Initialisieren:

```
       public FontCombo()
        {
            InitializeComponent();
```

Wir zeichnen die Einträge selbst:

```
            this.DrawMode = DrawMode.OwnerDrawFixed;
            this.ItemHeight = 28;
            this.DropDownStyle = ComboBoxStyle.DropDownList;
            this.DropDownWidth = 200;
            SetStyle(ControlStyles.Selectable, false);
        }
```

Zur Laufzeit werden die Fonts ermittelt und in der *Items*-Collection gespeichert:

```
        protected override void InitLayout()
        {
            base.InitLayout();
            this.Items.Clear();
            if (!this.DesignMode)
                foreach (FontFamily ff in (new InstalledFontCollection().Families))
                    try
                    {
                        Items.Add(new Font(ff, 12));
                    }
                    catch
                    { }
        }
```

Wir müssen uns um das Zeichnen eines Eintrag kümmern:

```
        protected override void OnDrawItem(DrawItemEventArgs e)
        {
```

Keine Auswahl:

```
            if (e.Index == -1)  return;
```

Hintergrund zeichnen:

```
            e.DrawBackground();
```

Auswahlrechteck zeichnen:

```
            if ((e.State & DrawItemState.Focus) != 0) e.DrawFocusRectangle();
```

String mit dem Namen der Schriftart und dem jeweiligen Font ausgeben:

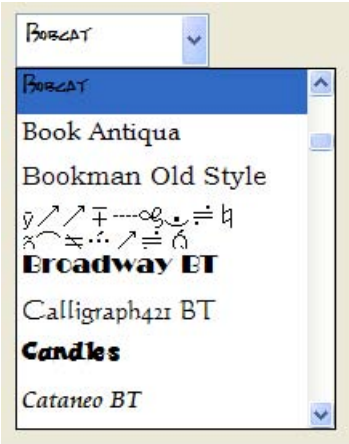
```
            e.Graphics.DrawString((Items[e.Index] as Font).Name,
                                  (Items[e.Index] as Font), Brushes.Black, e.Bounds);
        }
```

Über die Eigenschaft *SelectedFont* kann der Nutzer den ausgewählten Font abfragen und gleich nutzen:

```
        public Font SelectedFont
        {
            get
              {
                  if (this.SelectedIndex == -1)
                    return null;
                  else
                    return (this.SelectedItem as Font);
              }
        }
    }
}
```

Test

Erstellen Sie zunächst die Assembly und binden Sie dann die Komponente in den Werkzeugkasten ein. Ein kleines Testprogramm beweist die Funktionsfähigkeit:



Noch einmal: Das war ein **einfach** gehaltenes Beispiel, Sie können natürlich auch unterschiedliche Zeilenhöhen festlegen, die Stringlänge messen usw.

38.9.3 Das PropertyGrid verwenden

Sicher haben Sie sich auch schon gefragt, wie Sie zur Laufzeit die Eigenschaften von Objekten möglichst einfach und ohne großen Aufwand verändern können. Bevor Sie jetzt lange suchen, sollten Sie einen Blick auf die recht unscheinbare *PropertyGrid*-Komponente werfen, die schon zu .NET 1.x-Zeiten ein Aschenputteldasein in der Toolbox fristete.

Für unser Beispiel werden wir eine eigene Klasse definieren, deren Eigenschaften Sie per *PropertyGrid* zur Laufzeit bearbeiten können.

Oberfläche

Fügen Sie in ein Windows Formular eine *PropertyGrid*-Komponente ein. Das ist zunächst alles.

Quelltext

Zunächst definieren wir die gewünschte Klasse (Sie können das *PropertyGrid* aber auch an jede beliebige Komponente "klemmen").

```
class cMitarbeiter
{
    public cMitarbeiter(String Name, String Vorname, DateTime Geboren)
    {
        _Name = Name;
        _Vorname = Vorname;
```

```
            _geboren = Geboren;
            _Gehalt = 0;
        }

        private String _Name;
        [Category("Personaldaten")]
        public String Name
        {
            get { return _Name; }
            set { _Name = value; }
        }

        private String _Vorname;
        [Category("Personaldaten")]
        public String Vorname
        {
            get { return _Vorname; }
            set { _Vorname = value; }
        }

        private Single _Gehalt;

        [Category("Lohndaten")]
        public Single Gehalt
        {
            get { return _Gehalt; }
            set { _Gehalt = value; }
        }

        private int _Kinder;
        [Category("Lohndaten")]
        public int Kinder
        {
            get { return _Kinder; }
            set { _Kinder = value; }
        }
...
        private DateTime _geboren;
        [Category("Personaldaten")]
        public DateTime Geboren
        {
            get { return _geboren; }
            set { _geboren = value; }
        }
    }
```

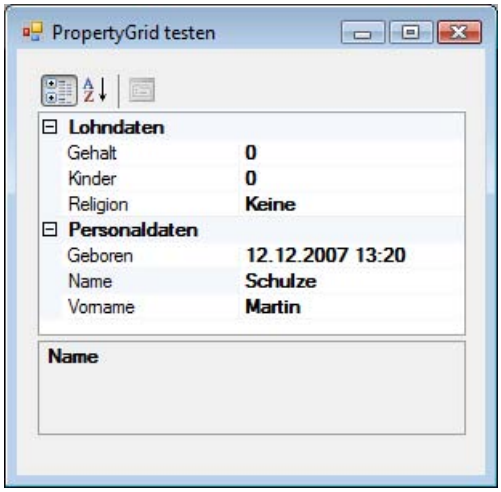
HINWEIS: Das Beispiel eignet sich auch recht gut, wenn Sie den Einfluss von Attributen testen wollen.

Die Verbindung von Grid und späterem Objekt ist absolut trivial:

```
private cMitarbeiter mitarbeiter = new cMitarbeiter("Schulze", "Martin", System.DateTime.Now);

private void Form1_Load(object sender, EventArgs e)
{
   propertyGrid1.SelectedObject = mitarbeiter;
}
```

Test

Beim Start der Anwendung sollten auch schon alle Eigenschaften des Objekts *mitarbeiter* angezeigt werden:



Teil VII: ASP.NET

- **Einführung in ASP.NET**
- **Übersicht ASP.NET-Controls**
- **Datenbindung unter ASP.NET**
- **ASP.NET-Objekte und Techniken**

Kapitel 39

Einführung in ASP.NET

In diesem und den folgenden Kapiteln beschäftigen wir uns mit einigen Aspekten der webbasierten Programmierung unter Visual Studio 2012. Der Schwerpunkt fällt dabei auf das Erstellen von ASP.NET-Anwendungen, den Pendants zu den Windows Forms-/WPF-Anwendungen.

39.1 ASP.NET für Ein- und Umsteiger

Haben Sie bereits früher Internet-Anwendungen entwickelt, ist Ihnen der Begriff *ASP* (*Active Server Pages*) sicher geläufig. Wenn nicht, dann finden Sie im folgenden Abschnitt einen kurzen Überblick.

39.1.1 ASP – Ein kurzer Blick zurück

Bei ASP handelt es sich um eine Umgebung für *serverseitige Skripte*, die zum Erstellen und Ausführen interaktiver Webanwendungen verwendet werden können. Besonders leicht haben es hier VB-Programmierer: Standard-Skriptsprache für ASP ist *VBScript*, ein enger Verwandter von Visual Basic. Wer mag, kann seine Skripte auch mit Microsoft *JScript*, *PERL*, *REXX* oder *Python* schreiben (die drei letzten Sprachen müssen über Drittanbieter bezogen werden).

Grundsätzlich handelt es sich bei jeder ASP-Datei um eine normale HTML-Datei, in die mittels spezieller Tags Skriptcode eingebettet ist.

BEISPIEL 39.1: Eine einfache ASP-Datei zur Anzeige von Datum und Uhrzeit

HTML

```
<HTML>
  <HEAD>
     <TITLE>Test</TITLE>
  </HEAD>
  <BODY>
    <%
    Dim uhrzeit
    Uhrzeit = DateAdd("h",9,Now)
    Response.Write "Aktuelle Zeit: " CStr(Uhrzeit)
```

BEISPIEL 39.1: Eine einfache ASP-Datei zur Anzeige von Datum und Uhrzeit

```
    %>
  </BODY>
</HTML>
```

Vielleicht fragen Sie sich, warum beim Datum mit *DateAdd* "gerechnet" wird? Die Antwort beschreibt auch gleich das Modell von ASP: Da die Skripte auf dem Web-Server (im Beispiel ein US-Server) ausgeführt werden, gilt für die *Time*-Funktion auch die Zeit des Servers. Soll die Seite in Deutschland korrekt angezeigt werden, müssen Sie die Zeitverschiebung berücksichtigen.

Der technische Ablauf ist recht einfach beschrieben: Wird durch den Browser (Client) eine *.asp*-Datei vom Server angefordert, durchsucht der Server die ASP-Seite nach Skriptcode. Dieser wird nachfolgend ausgeführt und das Endergebnis an den Client zurück gegeben. Der Client erhält also lediglich den reinen HTML-Quellcode der ursprünglichen ASP-Seite sowie die zusätzlich mittels Skript erzeugten HTML-Codes.

Für den Client handelt es sich bei der ASP-Seite um eine ganz normale HTML-Datei, in der sich im Regelfall kein Skriptcode mehr befindet. Damit brauchen Sie sich auch keine Gedanken um den Typ des Browsers zu machen, ASP-Dateien sollten mit **jedem** Browser korrekt angezeigt werden.

Dieser Vorteil und die Möglichkeit, Code vor dem Endverbraucher der Seite zu verbergen, sind die Vorteile gegenüber der clientseitigen Skriptprogrammierung.

39.1.2 Was ist bei ASP.NET anders?

ASP.NET ist die konsequente Weiterentwicklung von ASP. Geblieben ist lediglich das Grundprinzip, dass an den Client (fast nur) reiner HTML-Code gesendet wird, d.h., dass Sie sich um den jeweiligen Browser nur aus layouttechnischer Sicht zu kümmern brauchen.

Im Unterschied zum alten ASP wird bei ASP.NET auf dem Server eine eigenständige Assembly von der ASP.NET-Engine ausgeführt, die den erwünschten HTML-Code erzeugt. Es gibt keine Vermischung von HTML-Code und C#-Code[1]. Sie können ASP.NET-Anwendungen zunächst wie normale Windows-Programme in der Visual Studio-Umgebung entwickeln.

Allerdings sollten Sie beim Entwurf von ASP.NET-Anwendungen einen Grundsatz nie vergessen:

HINWEIS: Der von Ihnen programmierte C#-Code, d h. der Ereigniscode, wird immer auf dem Server ausgeführt. Der Client (Browser) erhält fast[2] reinen HTML-Code! Dies trifft auch bzw. gerade für Web Forms zu, die sich in der Entwicklungsumgebung wie ihre Windows Forms-Pendants verhalten.

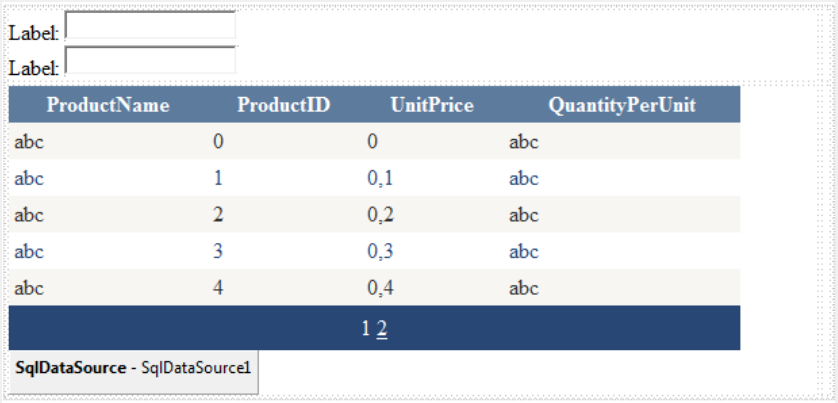
[1] Wer will kann auch mischen, aber das sollte man besser lassen.

[2] Mit der Einführung von AJAX wird auch Skriptcode übertragen. Wir kommen auf dieses Thema ab Seite 2089 zurück.

BEISPIEL 39.2: Entwurfs- und Laufzeitansicht

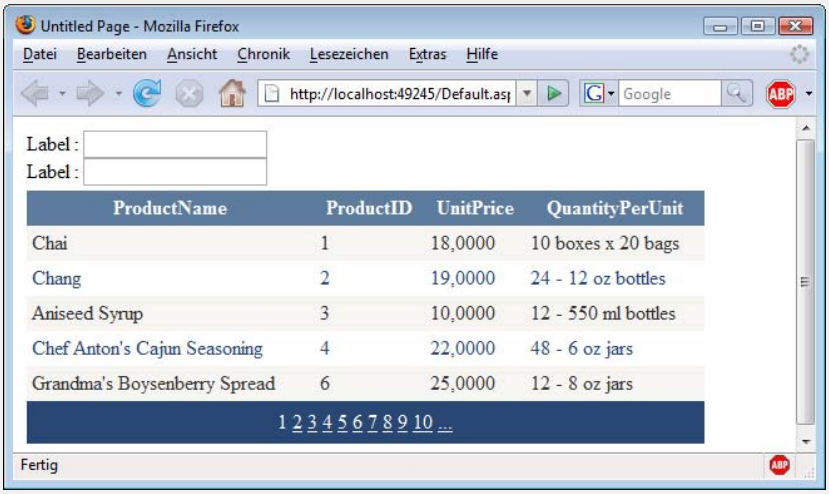
Entwurf

Web-Form in der Entwurfsansicht:



Laufzeitansicht

Aus Sicht des Clients (Browser):



HTML

Der dazugehörige Quellcode zeigt, dass es sich beim vermeintlichen Grid um eine einfache HTML-Tabelle handelt.

```
...
<table cellspacing="0" rules="rows" border="1" id="DataGrid1"
style="height:142px;width:640px;border-collapse:collapse;Z-INDEX: 101; LEFT: 13px; POSITION:
absolute; TOP: 74px">
 <tr style="color:Black;background-color:#CCFFFF;font-weight:bold;">
   <td colspan="12"><span>&lt;</span> 
   <a href="javascript:__doPostBack('DataGrid1:_ctl1:_ctl1','')"
       style="color:Black;">&gt;</a></td>
```

BEISPIEL 39.2: Entwurfs- und Laufzeitansicht

```
 </tr><tr style="background-color:Silver;font-weight:bold;"><td><a
href="javascript:__doPostBack('DataGrid1:_ctl2:_ctl2','')">au_fname</a></td><td><a
...
href="javascript:__doPostBack('DataGrid1:_ctl2:_ctl10','')">zip</a></td><td><a
href="javascript:__doPostBack('DataGrid1:_ctl2:_ctl11','')">contract</a></td>
 </tr><tr style="background-color:Yellow;">
...
<td>172-32-1176</td><td>White</td><td>Johnson</td><td>172-32-
1176</td><td>White</td><td>Johnson</td><td>408 496-7223</td>
<td>10932 Bigge Rd.</td><td>Menlo Park</td><td>CA</td><td>94025</td><td>True</td>
 </tr><tr style="background-color:Blue;">
```

Haben Sie sich obige Zeilen etwas genauer angesehen, wird Ihnen vielleicht aufgefallen sein, dass auch etwas Skriptcode zum Einsatz kommt. Haben aber die Autoren nicht ein paar Zeilen weiter oben geschrieben, dass der gesamte Code auf dem Server ausgeführt wird?

Die Antwort ist ja und nein. Der in den generierten Seiten enthaltene Skriptcode (JavaScript) dient lediglich dazu, Ereignisse auf dem Client an den Server **weiterzugeben**, wie uns das zugehörige Skript zeigt:

```
<script language="javascript">
<!--
 function __doPostBack(eventTarget, eventArgument) {
   var theform = document.Form1;
   theform.__EVENTTARGET.value = eventTarget;
   theform.__EVENTARGUMENT.value = eventArgument;
   theform.submit();
 }
// -->
</script>
```

ASP.NET bietet Ihnen auf dieser Basis die Möglichkeit, neben einfachen Anwendungen auch komplexe datenbankbasierte Applikationen zu erstellen, die über ein (fast) normales Ereignismodell verfügen. Doch Vorsicht:

HINWEIS: Wie gezeigt, müssen Client-Ereignisse an den Server weitergegeben werden, was eine intakte und vor allem schnelle Web-Verbindung erfordert.

39.1.3 Was gibt es noch in ASP.NET?

Neben den oben bereits vorgestellten Web Forms, d h. der Oberfläche Ihrer Webanwendungen, sind unter ASP.NET noch weitere Technologien/Begriffe von Interesse:

- **Web Server Controls**
  ... sind die "Steuerelemente" Ihrer Web Form-Anwendung. Aufgabe dieser Controls ist das Erzeugen (Rendern) von HTML-Ausgaben, die auf dem Client als Oberflächenelemente ange-

zeigt werden. Wie FCL-Controls sind auch Web Server Controls mit Eigenschaften, Methoden und Ereignissen ausgestattet.

- **User Controls**
  ... bieten eine Möglichkeit, Code und Oberflächenelemente bausteinartig wiederzuverwenden. User Controls lassen sich in Web Forms quasi wie Frames einblenden.
- **Web Services**
  ... sind auf dem Webserver angelegte Klassen, deren Methoden via Internet aufgerufen werden können. Daten bzw. Objekte werden von Webservices per SOAP in XML-Dokumenten verpackt und an den Client gesendet.
- **ADO.NET**
  ... bietet die Datenzugriffstechnologie, die um einige spezifische ASP.NET-Controls erweitert wurde (siehe Kapitel 40 und 41).
- **HTML Controls**
  ... die schon aus der ASP- bzw. HTML-Zeit bekannten Steuerelemente.
- **Ajax**
  ... bietet die Möglichkeit der asynchronen Datenübertragung zwischen Client (Browser) und Server. Dadurch wird es möglich, HTML-Seiten zu aktualisieren ohne sie komplett neu zu laden.
- **ASP.NET MVC**
  ... ist ein seit der Version 2010 auch in Visual Studio angebotenes Framework (**M**odel **V**iew **C**ontroller), das als Alternative zu den WebForms fungiert. Basierend auf einem Modell (Geschäftsobjekte, Datenzugriff etc.) werden unter Verwendung von Views die eigentlichen Endnutzerinterfaces erzeugt. Der Controller ist für die Interaktion zuständig.
- **Dynamic Data**
  Basierend auf einem Datenmodell wird eine komplette Website für die Interaktion generiert. Mit Hilfe spezieller Annotations können Sie die Gestaltung der Seiten beeinflussen.

39.1.4 Vorteile von ASP.NET gegenüber ASP

Gegenüber der früheren Verfahrensweise unter ASP bietet ASP.NET einige wesentliche Vorteile:

- Zunächst können Sie auf die unglückliche Verquickung von HTML- und Skriptcode verzichten, was der Übersichtlichkeit Ihrer Projekte und der Programmierung im Team wesentlich zugute kommt.
- Bedingt durch das ereignisgesteuerte Modell sind (hoffentlich) auch die Zeiten der Spagetti-Code-Programmierung vorbei. Die Sprache und der objektorientierte Ansatz zwingen zur übersichtlichen und leicht nachvollziehbaren Programmierung.
- Die ASP.NET-Komponenten erleichtern die Oberflächenprogrammierung wesentlich, in weiten Bereichen werden "Formulare" wie Windows-Formulare programmiert und entworfen. Die Entwicklung erfolgt mit Hilfe eines Formulardesigners, im besten Fall bekommen Sie keine

Zeile HTML-Code zu Gesicht, es kann auch vorkommen, dass Sie ganz auf C#-Code verzichten können.

- ASP.NET-Webpages bieten bereits ausgefeilte Grundgerüste für Routineaufgaben. So ist es beispielsweise problemlos möglich, dass sich ein und dieselbe Seite vom Server aufruft, um Statusänderungen in "Steuerelementen" anzuzeigen (*PostBack*). In ASP-Anwendungen war es meist einfacher, eine neue Seite aufzurufen.
- ASP.NET nutzt alle Vorteile des .NET-Frameworks, wie Inheritance, Sicherheit, Garbage Collections etc.
- ASP.NET-Anwendungen sind schneller als ASP-Anwendungen, da es sich um bereits kompilierten (dies kann auch temporär erfolgen) Code handelt.

Zum Schluss ein Wermutstropfen für alle, die noch über einen reichen Fundus an "alten" ASP-Seiten verfügen: Sie können diese zwar parallel zu den neuen ASP.NET-Seiten verwenden, ein Konvertieren dürfte jedoch in den meisten Fällen kaum den Aufwand rechtfertigen. Dies ist vor allem auch dem Datenmodell von ADO.NET geschuldet.

39.1.5 Voraussetzungen für den Einsatz von ASP.NET

Grundsätzlich können Sie, wenn es um die Voraussetzungen für den Einsatz von ASP.NET geht, davon ausgehen, dass es sich um serverseitige Anforderungen handelt.

Vorausgesetzt wird eines der aktuellen Microsoft Betriebssysteme wie Windows Server 2003/2008/2012 bzw. Windows Vista/7/8 (Business/Ultimate bzw. Pro/Enterprise).

Weiterhin müssen folgende Komponenten installiert sein:

- .NET Framework 4.5 SDK
- IIS 6 bzw. 7/7.5/8 (standardmäßig ist der IIS nicht installiert, Sie müssen dies über die Systemsteuerung nachholen)

Für den Entwurf der Anwendungen können Sie, müssen aber nicht, Visual Studio verwenden.

HINWEIS: Auf dem Entwicklungs-PC brauchen Sie nicht zwingend den IIS zu installieren, Visual Studio hat bereits einen eigenen Webserver "an Bord".

39.1.6 Und was hat das alles mit C# zu tun?

C# ist eine der Sprachen, mit denen Sie ASP.NET-Anwendungen entwickeln können, es könnte jedoch auch Visual Basic sein.

Haben Sie sich bisher als C#-Programmierer mehr oder weniger auf das Erstellen von Windows-Applikationen beschränkt, steht Ihnen mit ASP.NET auch das Internet/Intranet für Ihre Projekte zur Verfügung.

Grundsätzlicher Projektaufbau

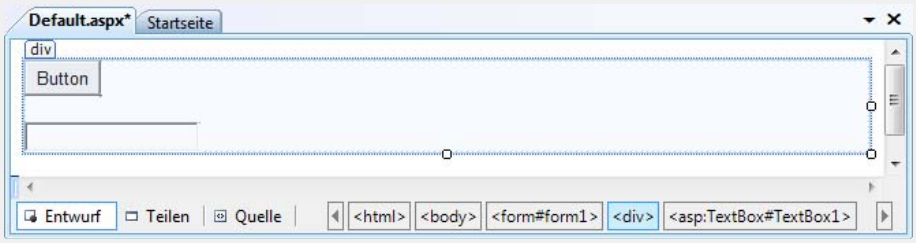
Natürlich könnten Sie Ihre ASP.NET-Projekte auch mit einem einfachen Texteditor programmieren, sowohl die Oberfläche als auch die dazugehörigen Klassen etc. sind reine Textdateien, bevor sie dynamisch kompiliert werden.

Visual Studio unterstützt Sie "lediglich" mit einem komfortablen Formular-Designer und diversen Tools, die Ihnen die Arbeit erleichtern sollen.

BEISPIEL 39.3: Formulardesigner/Quellcode

Oberfläche

Erstellen Sie in Visual Studio das folgende simple Web-Form



HTML

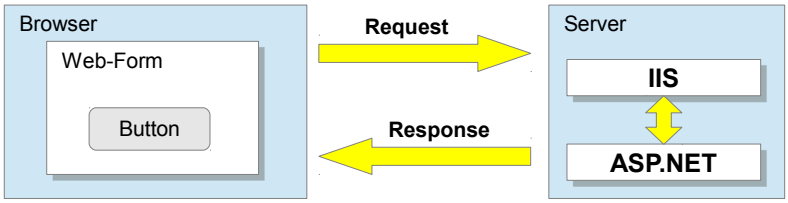
... wird im Hintergrund der folgende Code erzeugt (*Default.aspx*):

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="WebApplication1._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Untitled Page</title>
    </head>
<body>
    <form id="form1" runat="server">
    <br />
    <asp:Button ID="Button1" runat="server" Text="Button" />
    <br />
    <br />
    <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
    </form>
</body>
</html>
```

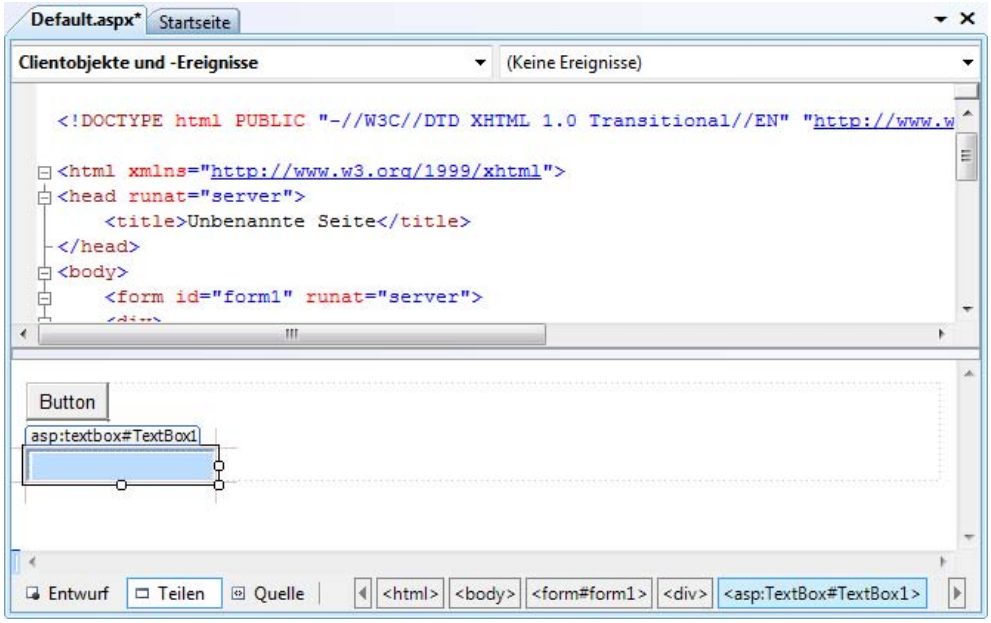
Wie Sie sehen, handelt es sich um reinen HTML-Code. Alle Komponenten befinden sich in einem Form-Block, um die Daten auch an den Server übertragen zu können.

Interessant ist auch das Attribut *"runat=server"*. Auch wenn Sie Code für das *Button_Click*-Ereignis geschrieben haben, finden Sie hier keinerlei Anweisungen, das Ereignis wird einfach an den

Server weitergereicht. Auf dem Server wird daraufhin eine Instanz der zugehörigen Klasse (siehe folgendes Beispiel) erzeugt und ausgeführt. Das Ergebnis dieses Aufrufs wird wiederum an den Client zurückgesendet.



Seit Visual Studio 2008 haben Sie die Möglichkeit, Design- und Source-Ansicht gleichzeitig darzustellen (wie auch beim WPF-Designer).



Allerdings ist es bei dieser Ansicht empfehlenswert, einen zweiten Bildschirm zu installieren und einige Visual Studio-Fenster auf diesen auszulagern, die eigentliche Arbeitsfläche tendiert andernfalls gegen null.

BEISPIEL 39.4: C#-Quellcode eines WebFormulars

C#

Die zum Formular gehörende Klasse, die vom Server ausgeführt wird (als C#-Programmierer dürfte Ihnen der Grundaufbau bereits geläufig sein):

```
using System;
using System.Collections;
using System.Configuration;
```

BEISPIEL 39.4: C#-Quellcode eines WebFormulars

```
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;

namespace WebApplication1
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {   }
```

Und hier finden wir dann auch die Ereignisbehandlung für das *Button1_Click*-Event (im vorliegenden Beispiel ändern wir einfach den Text des *TextBox*-Controls):

```
        protected void Button1_Click(object sender, EventArgs e)
        {
            TextBox1.Text = "Hallo World";
        }
    }
}
```

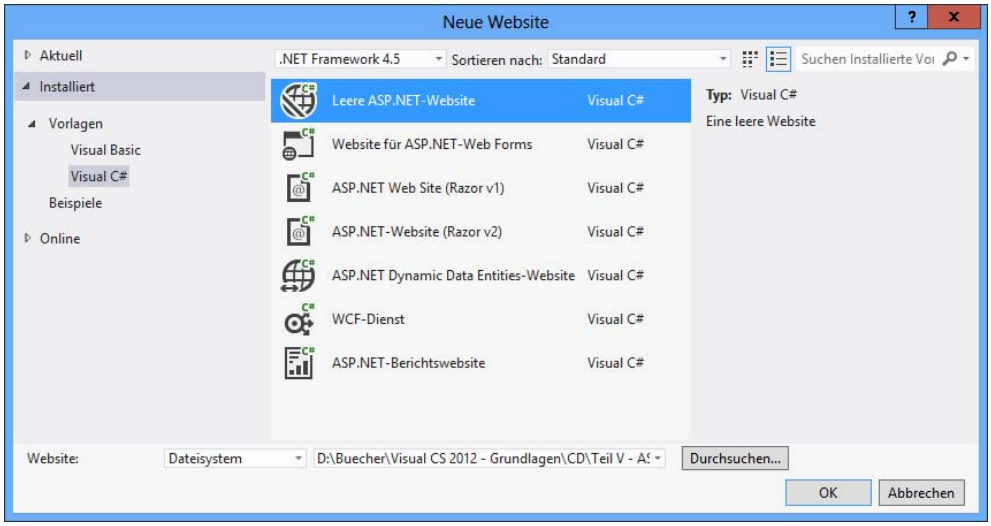
Damit haben Sie auch schon einen ersten Blick auf die strikte Trennung von Code und Oberfläche geworfen und zwei Dateitypen (*Default.aspx* und *Default.aspx.cs*) kennen gelernt. Was es mit den anderen Dateien des Projekts auf sich hat, erfahren Sie in den folgenden Abschnitten am praktischen Beispiel.

39.2 Unsere erste Web-Anwendung

Das Prinzip der objekt- und ereignisorientierten Programmierung gilt in Visual Studio uneingeschränkt auch für Web-Anwendungen, wie unser erstes Beispiel – ein kleiner Währungsrechner – zeigen soll.

39.2.1 Visueller Entwurf der Bedienoberfläche

Eine neue ASP.NET-Website erstellen Sie, wenn Sie zunächst *Datei|Neu|Webseite* wählen. In der Liste der für C# angebotenen Vorlagen wählen Sie *Leere ASP.NET-Website*:

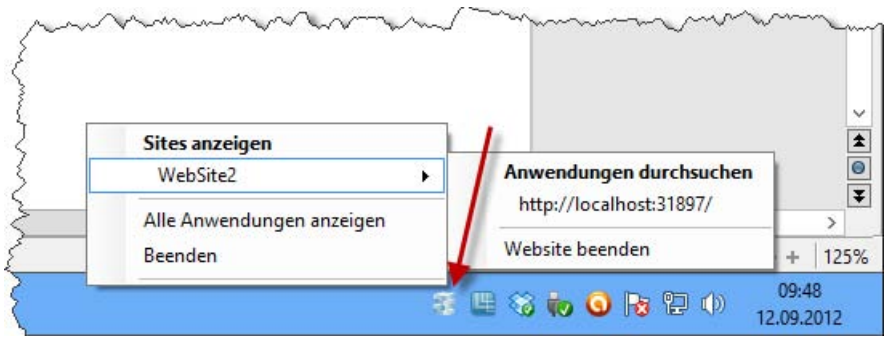


Geben Sie einen neuen Verzeichnisnamen an, z.B. *EuroDollar*. Mit der Option *Speicherort* können Sie entscheiden, ob Ihr Projektmappen-Explorer

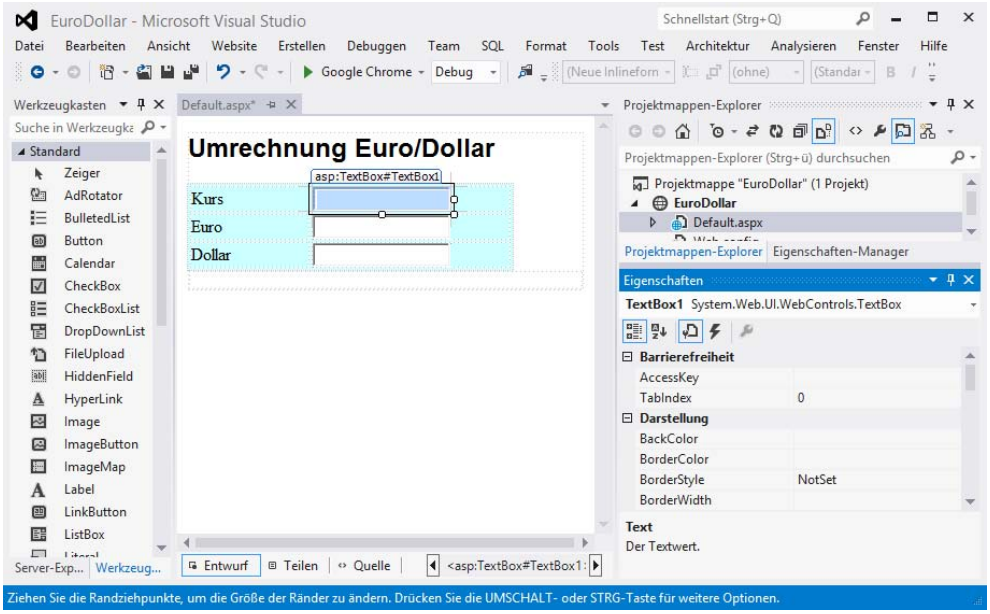
- als Unterverzeichnis im Dateisystem,
- per HTTP auf einem IIS oder
- per FTP-Zugriff

erzeugt wird. Wir belassen es in der Entwurfsphase beim *Dateisystem*, das Projekt kann später problemlos auf einen IIS exportiert werden.

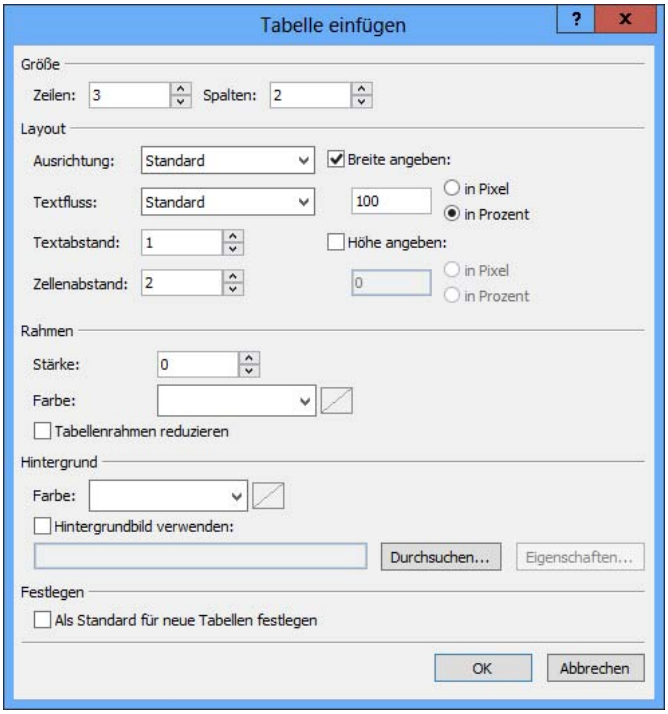
HINWEIS: Haben Sie die Option *Dateisystem* gewählt, können Sie das Projekt nur aus Visual Studio heraus testen. Dazu wird ein eigener Webserver (IIS-Express) verwendet, ein entsprechendes Icon ist beim Start der Anwendung im Taskbar zu sehen.



Nach kurzer Wartezeit erscheint die Web-Entwicklungsumgebung von Visual Studio, welche sich kaum von der IDE für Windows-Anwendungen abhebt (Die Seite *Default.aspx* müssen Sie erst über *Website|Neues Element ... WebForm* hinzufügen.).



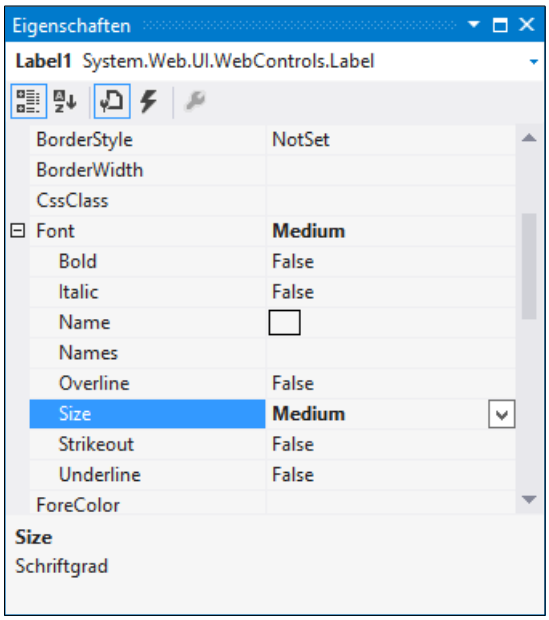
Erstellen Sie nun die oben abgebildete Web-Bedienoberfläche. Verwenden Sie nur die Steuerelemente aus der Werkzeugkasten-Kategorie "Standard". Die erforderliche HTML-Tabelle lässt sich über den Menüpunkt *Tabelle|Tabelle einfügen* erzeugen:



Beim Vergleich mit einer Windows Forms-Oberfläche fällt z.B. das Fehlen der Formular-Titelleiste und eines *Beenden*-Buttons auf. Da eine Web-Anwendung aber nicht in Windows-Formularen, sondern im Internet Explorer angezeigt wird, gibt es für beides keine Notwendigkeit mehr. Quasi als Ersatz für die fehlende Titelleiste haben wir oben einen reinen HTML-Text positioniert.

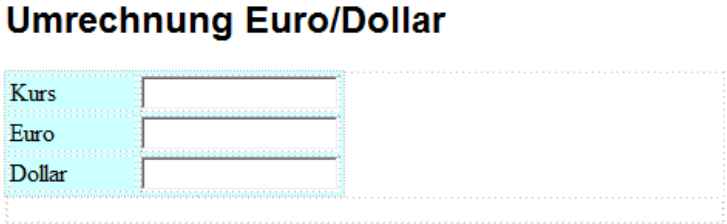
39.2.2 Zuweisen der Objekteigenschaften

Beginnen wir ganz oben mit dem Titeltext. Als rein kosmetische Korrektur haben wir die Schriftgröße geändert. Nachfolgend können Sie die Beschriftung (*Text*) und die Schriftattribute der drei *Label* anpassen.



Es ist typisch für Web-Anwendungen, dass es keine festen Pixel-Größen für die Schriftgröße mehr gibt, sondern lediglich allgemeine Angaben wie *Small*, *Medium* und *Large* (siehe Abbildung).

Nach dem Zuweisen der Eigenschaften für die übrigen Steuerelemente, bei denen Sie ruhig etwas experimentieren können, sollte sich etwa die im Folgenden abgebildete Oberfläche ergeben:



Und zum Schluss bitte nicht vergessen:

HINWEIS: Setzen Sie die *AutoPostBack*-Eigenschaft der drei *TextBox*en auf *True*, wir kommen auf die Bedeutung später zurück.

39.2.3 Verknüpfen der Objekte mit Ereignissen

Wechseln Sie über den Projektmappen-Explorer zur Code-Ansicht *Default.aspx.cs* (bitte nicht mit der HTML-Ansicht verwechseln!). Abgesehen von einigen wenigen Unterschieden ist die Programmierung identisch mit der von Windows Forms-Anwendungen.

Der Rahmencode des Event-Handlers für das *Load*-Ereignis der Web-Form ist bereits vorgefertigt[1]. Unmittelbar zuvor deklarieren wir die benötigten Variablen. Beim Laden der Seite werden den Variablen Anfangswerte zugewiesen[2] und die Textboxen damit initialisiert:

```
using System;
...
using System.Xml.Linq;

public partial class _Default : System.Web.UI.Page
{
    private Single euro, dollar, kurs;

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            euro = 1;
            dollar = 1;
            kurs = 1;
            TextBox1.Text = euro.ToString();
            TextBox2.Text = dollar.ToString();
            TextBox3.Text = kurs.ToString();
        }
    }
}
```

HINWEIS: Obige Abfrage der *IsPostBack*-Eigenschaft des Web-Formulars bewirkt, dass nur beim ersten Aufruf der Seiteninhalt aktualisiert wird und nicht bei jedem Neuaufbau der Seite.

[1] Wenn nicht, einfach doppelt auf das WebForm klicken.

[2] Auf das Initialisieren der globalen Variablen bereits beim Deklarieren müssen wir leider verzichten, da sich dies sonst bei jedem Neuaufbau der Seite wiederholen würde. Es erfolgt also nur die einfache Deklaration.

Nun müssen wir die Event-Handler für die drei *TextBox*en erstellen. Da die für Web-Forms zuständigen Steuerelemente auf deutlich weniger Ereignisse als ihre Windows Forms-Pendants reagieren, gibt es u.a. auch kein *KeyUp*-Event[1]. Stattdessen werten wir das *TextChanged*-Ereignis aus.

Der Rahmencode wird auf uns bereits bekannte Weise erstellt: Einfach in der Ereignisliste des Eigenschaften-Fensters auf das *TextChanged*-Event doppelklicken.

Die Euro-Änderung:

```
protected void TextBox2_TextChanged(object sender, EventArgs e)
{
    euro = Convert.ToSingle(TextBox2.Text);
    kurs = Convert.ToSingle(TextBox1.Text);
    dollar = euro * kurs;
    TextBox3.Text = dollar.ToString("#,##0.00");
}
```

Die Dollar-Änderung:

```
protected void TextBox3_TextChanged(object sender, EventArgs e)
{
    dollar = Convert.ToSingle(TextBox3.Text);
    kurs   = Convert.ToSingle(TextBox1.Text);
    euro   = dollar / kurs;
    TextBox2.Text = euro.ToString("#,##0.00");
}
```

Die Kurs-Änderung:

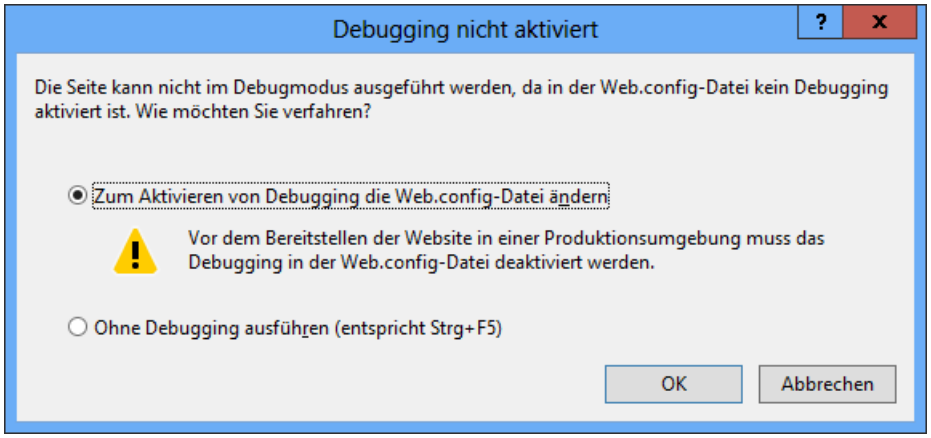
```
protected void TextBox1_TextChanged(object sender, EventArgs e)
{
    kurs = Convert.ToSingle(TextBox1.Text);
    euro = Convert.ToSingle(TextBox2.Text);
    dollar = euro * kurs;
    TextBox3.Text = dollar.ToString("#,##0.00");
}
```

39.2.4 Programm kompilieren und testen

Nach dem Start (F5) werden Sie zunächst mit der in der folgenden Abbildung gezeigten Dialogbox konfrontiert.

Bestätigen Sie einfach mit *OK*, um eine neue *Web.config* zu erzeugen und im Debug-Modus fortzufahren.

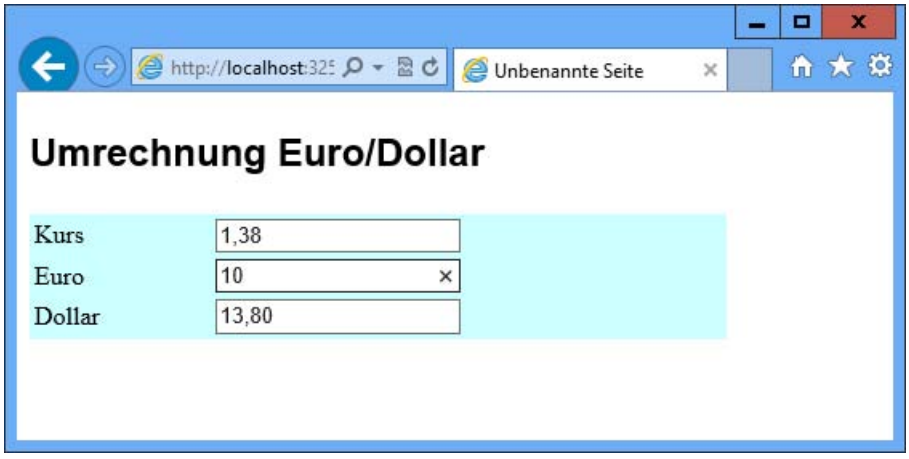
[1] Das ist ja eigentlich auch logisch, denn der Netzwerk-Traffic wäre gewaltig.



Nach einiger Zeit erscheint der Browser und die Seite mit der Adresse

```
http://localhost:32583/Default.aspx
```

wird anzeigt. Sie werden feststellen, dass die Textboxen nicht sofort aktualisiert werden, sondern erst nachdem eine Textbox den Fokus verliert, d h., wenn Sie mit der Maus in eine andere Textbox klicken bzw. die *Tab*-Taste betätigen (siehe unten).



39.3 Die ASP.NET-Projektdateien

Im Folgenden wollen wir – nach einem hoffentlich erfolgreich verlaufenen Test – die ASP.NET-Projektdateien näher beleuchten.

Im Gegensatz zu einer Windows Forms-Anwendung benötigt ein ASP.NET-Website-Projekt keine Projektdatei, da jede einzelne ASPX-Datei als eigene Assembly betrachtet wird. So besteht auch die Möglichkeit, in jedem Web Form eine andere Sprache zu verwenden, was der Übersichtlichkeit aber sicher nicht zuträglich ist.

39.3.1 Die ASP.NET-Projekttypen

Wenn wir uns schon über ein "ASP.NET-Website-Projekt" unterhalten, so ist es sicher auch sinnvoll, gleich einen Blick auf die unterschiedlichen Projekttypen bei ASP.NET-Anwendungen zu werfen:

Bereits über den Menüpunkt *Datei|Neu|Website* eröffnet sich ein breites Angebot:

- **Leere ASP.NET-Website**
- **Website für ASP.NET-Web Forms**
- ASP.NET Web Site (Razor v1)
- ASP.NET Web Site (Rasor v2)
- ASP.NET Dynamic Data Enities-Website
- WCF-Dienst
- ASP.NET-Berichtswebsite

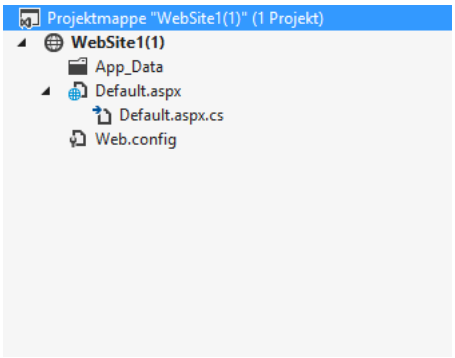
Noch dicker kommt es über den Menüpunkt *Datei|Neu|Projekt* (Rubrik *Internet*):

- **Leere ASP.NET-Webanwendung**
- **ASP.NET-Web Forms-Anwendung**
- ASP.NET MVC 3 Webanwendung
- ASP.NET MVC 4 Webanwendung
- Webanwendung für ASP.NET Dynamic Data Enities
- ASP.NET AJAX-Serversteuerelement
- ASP.NET AJAX-Serversteuerelement-Extender
- ASP.NET-Serversteuerelement

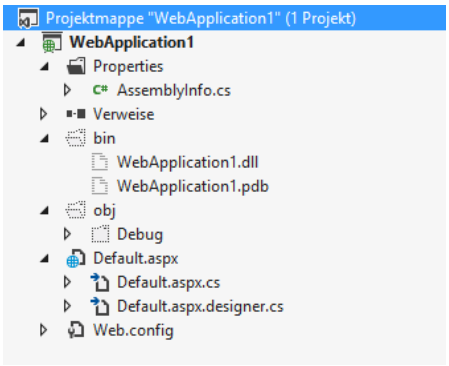
HINWEIS: Im Weiteren werden wir uns nur mit den oben fett hervorgehobenen Projekttypen beschäftigen.

Grundsätzlich scheint es zwei Projekttypen zu geben, die sich **...-Website** oder **...-Webanwendung** nennen. Worin liegen die Unterschiede? Auf den ersten Blick werden Sie kaum etwas feststellen, erst ein Blick in den Projektmappen-Explorer verrät uns mehr.

ASP.NET-**Website**:



ASP.NET-**Webanwendungen**:



ASP.NET-Website

Die ASP.NET-Website stellt sich als recht loser Verbund einzelner ASPX-Dateien mit zugehörigen ASPX.CS-Quellcode-Dateien dar, die im Normalfall erst beim Zugriff durch den Web-Client in einzelne Assemblies kompiliert werden.

Diese finden Sie beispielsweise im Verzeichnis (ASP.NET 4.5)

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files\
```

Als Folge dieser Vorgehensweise können Sie jederzeit die Oberflächenbeschreibung (ASPX) oder den Quellcode (*.aspx.cs*) ändern. Beim nächsten Seitenaufruf durch den Client wird der ASP.NET-Compiler die betreffende Seite neu kompilieren. Der Nachteil dieser Verfahrensweise: große Projekte werden schnell unübersichtlich und ganz nebenbei liegen die Quellcodes des Projekts für (fast) jeden einsehbar auf dem Server herum.

Ein weiterer Nachteil ist der Austausch von Informationen zwischen den eigentlich autarken Assemblies. Aus diesem Grund werden in Website-Projekten gemeinsame Codeteile (z.B. Klassen) im Unterordner *App_Code* gespeichert. Nur in diesem Fall können alle Seiten auf diese Definitionen zugreifen.

ASP.NET-Web-Anwendungen

Demgegenüber werden ASP.NET-Web-Anwendungen bereits durch Visual Studio in **eine** Assembly kompiliert, diese wird im *\bin*-Verzeichnis abgelegt. Da es sich um eine spezifische Assembly handelt, können Sie auch per *AssemblyInfo.cs* Einfluss auf Attribute und Eigenschaften nehmen. Im Fall einer Distribution genügt es, wenn Sie die ASPX-Dateien und die Assembly auf das Zielsystem kopieren[1].

Diese Projekttyp dürfte am besten den Desktop-Anwendungen entsprechen und damit für professionelle Projekte geeignet sein.

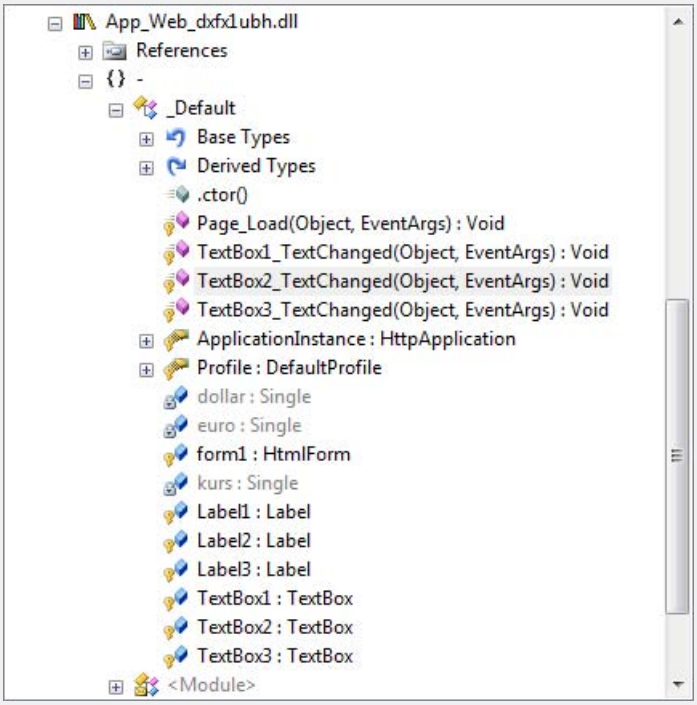
[1] Es ist auch möglich, alle Dateien in eine Assembly zu kopieren (*Web Deployment Project*).

39.3.2 ASPX-Datei(en)

Wie unser Einstiegsbeispiel schon gezeigt hat, besteht Ihre Webanwendung unter anderem auch aus einzelnen ASPX-Dateien (z.B. *Default.aspx*). Diese enthalten zum einen die Informationen über den Oberflächenaufbau der einzelnen Web-Formulare, zum anderen findet sich hier die Referenz auf den Ereigniscode bzw. die Formularklasse, die später in einer extra Formular-Assembly abgelegt wird.

HINWEIS: Da "normale" ASP.NET-Projekte erst bei Bedarf, d h. zur Laufzeit bzw. beim ersten Aufruf, kompiliert werden, ist die Suche nach den erzeugten Assemblies recht mühsam.

BEISPIEL 39.5: Web Form-Assembly für das Einstiegsbeispiel (im .NET-Reflector)



Für die aktuelle ASP.NET-Version und das vorliegende Projekt finden sich diese im Ordner

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET Files\eurodollar ...
```

HINWEIS: Für das Ziel-Framework 3.5 und früher finden Sie die Dateien im entsprechenden Ordner von .NET 2.x!

Doch zurück zu obigen Assemblies. Der ASP.NET-Compiler erzeugt diese immer bei Änderungen in der Code-behind-Datei (*.aspx.cs*). Sie können dies anhand des Dateidatums jederzeit überprüfen.

Die eigentlichen ASPX-Dateien sind HTML-Dateien, wie ein Blick in die Codeansicht (nicht **.cs*) verrät:

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
    <style type="text/css">
        .style1
...
    </style>
</head>
<body>
    <p class="style3">
        Umrechnung Euro/Dollar</p>
    <form id="form1" runat="server">
    <table class="style1">
        <tr>
            <td class="style2">
                <asp:Label ID="Label1" runat="server" Text="Kurs"></asp:Label>
            </td>
            <td>
                <asp:TextBox ID="TextBox1" runat="server" AutoPostBack="True"
                    ontextchanged="TextBox1_TextChanged"></asp:TextBox>
            </td>
        </tr>
...
    </table>
    <div>
    </div>
    </form>
</body>
</html>
```

In der Visual Studio-IDE können Sie sowohl die grafische Design-Ansicht als auch die Codeansicht mit dem automatisch generierten HTML-Code aufrufen.

Fügen Sie in der Design-Ansicht zum Beispiel eine *TextBox* (Kategorie *Standard*) ein, erzeugt Visual Studio daraus folgenden HTML-Quellcode:

```
<asp:TextBox ID="TextBox4" runat="server" Height="65px" Width="199px"></asp:TextBox>
```

Für den HTML-erfahrenen Programmierer ist dies auf den ersten Blick etwas Neues, beachten Sie jedoch, dass dieses Element noch auf dem Server interpretiert werden muss, bevor es zum Client gesendet wird.

Das Resultat nach der Interpretation durch die ASP.NET-Engine ist altbekannt:

```
<input name="TextBox4" type="text" id="TextBox4" style="height:65px;width:199px;" />
```

Alle im Web Form platzierten Komponenten werden wiederum in ein Form-Element eingebettet:

```
<form runat="server">
  <asp:label id=Label1
   ...
</form>
```

Die Frage bleibt allerdings, wie die ASP.NET-Engine den Zusammenhang zwischen dem obigen "ASP.NET-Quellcode" und Ihrem Quellcode, d h. der Visual Studio-Assembly, herstellt.

Ein Blick an den Anfang des ASPX-Dokuments zeigt die Lösung. Mit der *Page*-Direktive werden diverse Attribute definiert:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
...
```

Mit Hilfe des *CodeFile*-Attributes stellt ASP.NET zunächst den Zusammenhang zwischen Formular und C#-Quellcode (Formular-Klasse) her.

HINWEIS: Diese Information wird **nicht** zur Laufzeit genutzt!

Die wesentliche Info für die ASP.NET-Engine ist das *Inherits*-Attribut, mit dem ein Zusammenhang zwischen ASP.NET-Formular und zugehöriger Klasse hergestellt wird. Die Engine braucht nur noch im untergeordneten /*bin*-Verzeichnis (dieses finden Sie nur, wenn Sie das Projekt vorkompilieren) oder im bereits beschriebenen ASP.NET-Temp-Verzeichnis nach der Assembly zu suchen und die entsprechende Klasse zu instanziieren.

Wie weit die Trennung von ASP.NET-Code und C#-Code geht, können Sie durch das direkte Verändern der ASPX-Datei ausprobieren. Verändern Sie einfach die Größe der Controls. Sie werden feststellen, dass ein erneutes Kompilieren der Assembly so lange nicht erforderlich ist, wie Sie keine Controls hinzufügen oder löschen.

39.3.3 Die aspx.cs-Datei(en)

Hier findet sich Ihr eigentliches Wirkungsfeld als Programmierer. Strukturell gleicht eine Web-Form-Klasse der entsprechenden Windows Forms-Klasse, allerdings haben wir es hier mit einem anderen Vorfahren und einem etwas anderen Ereignismodell zu tun.

```
using System;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
```

```
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {

    }
}
```

Quellcode zum Erzeugen von Komponenten und deren Konfiguration, wie bei den Windows Forms, werden Sie hier allerdings nicht finden. Wie bereits im vorhergehenden Abschnitt gezeigt, ist dies Angelegenheit der ASPX-Dokumente.

Auch das Verbinden von Oberflächenelementen mit dem Quellcode, d.h. das Zuordnen von Ereignishandlern, wird direkt im ASPX-Quellcode vorgenommen. Dort müssen Sie auch die Zuordnung löschen, wenn Sie mal eine Ereignismethode entfernt haben und die Zuordnung noch besteht:

```
<asp:TextBox ID="TextBox3" runat="server" AutoPostBack="True"
             OnTextChanged="TextBox3_TextChanged"></asp:TextBox></td>
```

39.3.4 Die Datei Global.asax

Wer jetzt nach einem Hauptprogramm o.ä. sucht, wird zunächst nicht fündig werden. Die gesuchte Funktionalität lässt sich jedoch meist mit einer globalen Anwendungsklasse realisieren. Diese müssen Sie allerdings erst dem Web hinzufügen (*Website|Neues Element hinzufügen*).

Diese Klasse implementiert diverse Ereignishandler, über die Sie auf den Start Ihrer Anwendung reagieren und zum Beispiel gemeinsame Ressourcen initialisieren. Doch Achtung: Webanwendungen verhalten sich naturbedingt anders als lokale Windows Forms-Anwendungen. Ihre Webanwendung kann gleichzeitig von hunderten Nutzern verwendet werden. So müssen Sie zwischen

- userspezifischen Daten (*Session*) und
- anwendungsglobalen Daten (*Application*) unterscheiden.

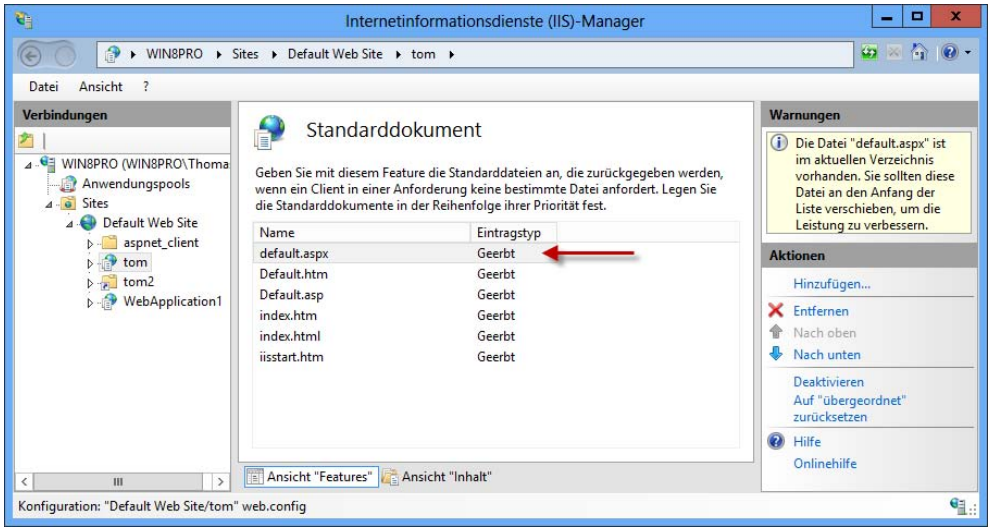
HINWEIS: Mehr zu dieser Thematik finden Sie ab Seite 2039.

39.3.5 Das Startformular

Im Gegensatz zu einer Windows Forms-Anwendung hat eine Web Form-Anwendung kein direktes Startformular, das zum Beispiel über die Projektdatei festgelegt wird. Entscheidend ist, welche ASPX-Datei zuerst im Browser des Endanwenders aufgerufen wird.

Ergibt sich der Aufruf der "Startseite" durch den Link eines übergeordneten Webs, dürften Sie kaum Schwierigkeiten haben. Soll jedoch gleich die erste Seite Ihres Webs eine ASPX-Seite sein, müssen Sie dies zum Beispiel im IIS-Manager entsprechend festlegen.

Wie auch bei den HTML-Seiten, bei denen Sie eine *Default.htm* o.ä. bestimmen konnten, definieren Sie einfach Ihre Startseite als *Default.aspx* und legen diese als Standarddokument für Ihr Web fest (siehe folgende Abbildung am Beispiel des IIS 8).



Jetzt genügt zum Beispiel auch der Aufruf

```
http://localhost/EuroDollarWeb
```

statt

```
http://localhost/EuroDollarWeb/WebForm1.aspx
```

39.3.6 Die Datei Web.config

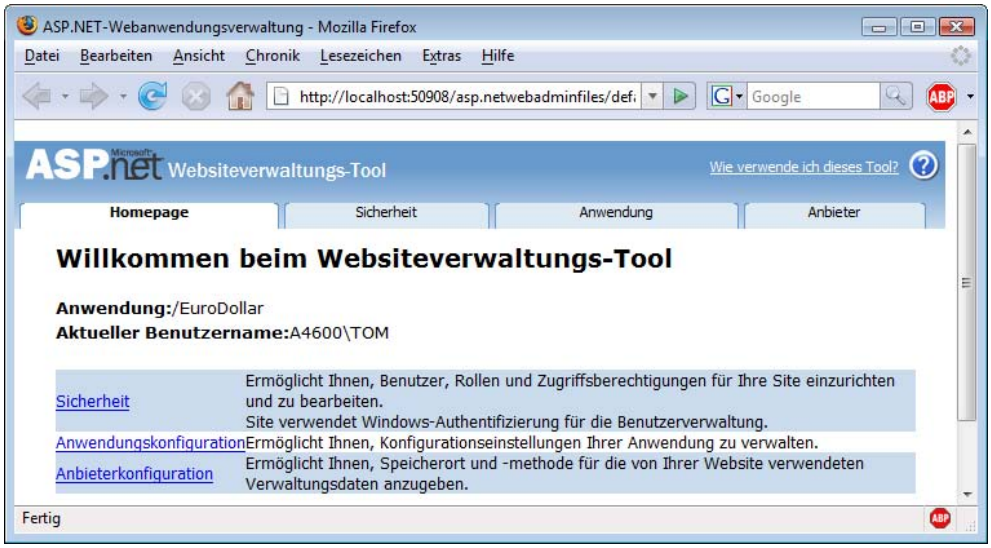
Werfen Sie einen Blick in die Projektverwaltung, so finden Sie neben den bereits besprochenen Dokumenten auch eine *Web.config*-Datei.

HINWEIS: Haben Sie ein leeres ASP.NET-Projekt gewählt, ist dieses Datei noch recht übersichtlich, dies ändert sich erst, wenn auch Anmeldeinformationen, Connectionsstrings etc. hier abgelegt werden.

Diese zu jedem ASP.NET-Projekt gehörende Datei dient der Konfiguration der aktuellen Anwendung. Ein Blick hinein verrät uns, dass es sich um XML-Daten handelt, die bereits mit reichlich Kommentaren angereichert sind:

```
<?xml version="1.0"?>

<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->

<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated Security=SSPI;AttachDBFilename=|DataDirecto
         providerName="System.Data.SqlClient" />
  </connectionStrings>

  <system.web>
    <compilation debug="false" strict="false" explicit="true" targetFramework="4.0" />

    <authentication mode="Forms">
      <forms loginUrl="~/Account/Login.aspx" timeout="2880" />
    </authentication>

    <membership>
      <providers>
        <clear/>
        <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembershipProvider" connectio
             enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestionAndAnswer="false"
             maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6" minRequiredNonalphanumericCharac
             applicationName="/" />
      </providers>
    </membership>
```

HINWEIS: Statt sich mit kryptischen XML-Tags herumzuschlagen, können Sie viele der in der Datei *web.config* vorgenommenen Einstellungen auch über den Menüpunkt *Website|ASP.NET-Konfiguration* verändern.



Ein gewisses Grundverständnis für den Inhalt der *Web.Config* sollten Sie aber auf jeden Fall besitzen, deshalb hier in Kürze die wichtigsten Einträge bzw. deren Aufgaben:

- *<appSettings>*
  Verwaltung von Anwendungseinstellungen
- *<authentication mode="Windows|Forms|Passport|None|Federated">*
  Konfiguration der Authentication, d h. des Identifizierens von Usern
- *<authorization>*
  Konfiguration der Authorization, d h. die Verwaltung von Rechten
- *<compilation>*
  Konfiguration des .NET Framework Compilers
- *<cache>*
  Caching-Konfiguration
- *<customErrors>*
  Einstellungen für das Verhalten im Fehlerfall
- *<globalization>*
  Lokalisierungseinstellungen
- *<identity>*
  kontrolliert die Identität der Web-Anwendung (Impersonation)
- *<machineKey>*
  verwaltet die Schlüssel für Ver-/Entschlüsselung
- *<mailSettings>*
  Mail-Server-Konfiguration, wird von *MailClient* verwendet
- *<pages>*
  seitenspezifische Einstellungen für alle Formulare der Anwendung, diese können auf Seitenebene überschrieben werden
- *<processModel>*
  konfiguriert das ASP.NET-Prozessmodell
- *<sessionState>*
  konfiguriert die Session-Informationen
- *<trace>*
  konfiguriert die Trace-Informationen für die Anwendung
- *<webServices>*
  kontrolliert die Einstellungen für XML Web Services
- *<configuration>*
  fügt eigene Einstellungen hinzu, die Sie mit Ihrer Web-Anwendung auslesen können

Welche Optionen im Einzelnen möglich sind, zeigen die weiteren Abschnitte des Kapitels im Zusammenhang mit bestimmten Aufgabenstellungen.

HINWEIS: Möchten Sie innerhalb Ihrer IIS-Anwendung verschiedene Konfigurationen realisieren, kann in jedem Unterverzeichnis eine eigene *Web.config* angelegt werden.

Dies kann auch zur Laufzeit des Webs geschehen, jede Änderung der *Web.config*-Datei wird vom Server überwacht und entsprechend umgesetzt. Ein Neustart des IIS oder der Web-Anwendung ist in diesem Fall nicht erforderlich.

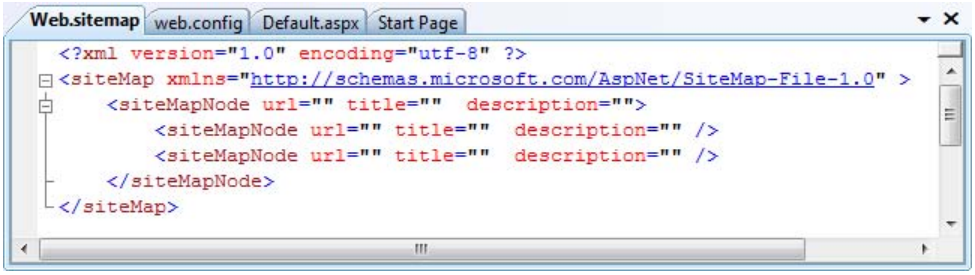
39.3.7 Masterpages (master-Dateien)

Sollten Sie über eine *.master*-Datei gestolpert sein, handelt es sich quasi um eine Vorlage für weitere Web Forms, die alle das gleiche Layout aufweisen.

HINWEIS: Mehr zum Thema finden Sie ab Seite 1950.

39.3.8 Sitemap (Web.sitemap)

Diese Datei stellt das "Inhaltsverzeichnis" Ihrer Webanwendung dar. Über geschachtelte *siteMapeNode*-Elemente kann der hierarchische Zusammenhang zwischen den einzelnen Seiten hergestellt werden (leider nur von Hand).

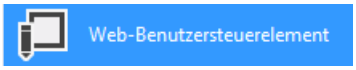


ASP.NET nutzt diese Datei bei einigen Navigations-Controls (*TreeView*, *Menu*) für die automatische Darstellung.

HINWEIS: Mehr zum Thema finden Sie ab Seite 1938.

39.3.9 Benutzersteuerelemente (ascx-Dateien)

Benutzersteuerelemente/User-Controls stellen einen Zusammenfassung mehrerer einzelner Controls dar. Diese lassen sich mit einem Interface versehen und in Webseiten einbauen.



HINWEIS: Mehr Informationen zum Programmieren von Web-Benutzersteuerelementen finden Sie ab Seite 2095.

39.3.10 Die Web-Projekt-Verzeichnisse

Ausgehend vom Stammverzeichnis Ihrer Webanwendung nutzt ASP.NET noch einige spezielle Unterverzeichnisse, von denen Ihnen sicher *App_Data* bereits aufgefallen ist. Eine Übersicht finden Sie in der folgenden Tabelle:

Verzeichnis	Bedeutung
App_Data	In diesem Verzeichnis werden die Daten(banken) Ihrer Anwendung abgelegt. Dies können neben Access- (*mdb*) und SQL Serverdaten-banken (*mdf*) auch XML-Dateien sein. In diesem Verzeichnis werden auch die User/Rollen-Informationen für die Forms-Authentication gespeichert. Ein direkter Download von Daten aus diesem Verzeichnis wird von ASP.NET verhindert.
App_Code	Hier wird gegebenenfalls der Quellcode für gemeinsame Klassen abgelegt. Diese sind dann in allen WebForms verfügbar.
App_Browsers	Hier können Browserdefinitionen abgelegt werden, die von ASP.NET zur Browser-Identifikation genutzt werden.
App_GlobalResources	Speichern Sie hier globale Ressourcen Ihrer Anwendung.
App_LocalResources	Speichern Sie hier spezifische Ressourcen für Benutzersteuerelemente etc.
App_Themes	Enthält weitere Unterverzeichnisse, in denen Ressourcen (Skins, StyleSheets, Grafiken etc.) für einzelnen Themes enthalten sind.
App_WebReferences	Handelt es sich um eine Webservice-Anwendung, werden in diesem Verzeichnis Verweisvertragsdateien, Schemas und Discovery-Dokumentdateien abgelegt.
bin	Nach dem Kompilieren der Anwendung enthält das Verzeichnis die kompilierten Assemblies für Steuerelemente, Web Forms etc.

39.4 Lernen am Beispiel

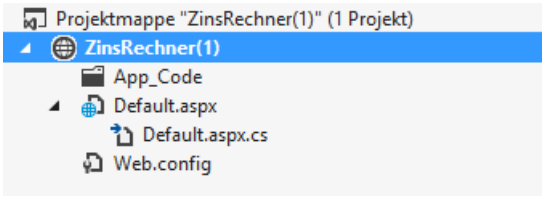
Wir wollen am Beispiel einer Zinseszins-Berechnung die wichtigsten Etappen des ASP.NET-Projektentwurfs ansprechen. In den folgenden Abschnitten binden wir dann weitere Komponenten ein, um unser Programm – um etwas anderes handelt es sich ja nicht – komfortabler zu machen.

HINWEIS: Bitte verzweifeln Sie nicht, wenn zwischendurch teilweise auf andere ASP.NET-Kapitel verwiesen wird, um Details zu einzelnen Aspekten zu beleuchten.

39.4.1 Erstellen des Projekts

Erstellen Sie ein neues Web-Projekt (*Leere ASP.NET-Website*) über den Menüpunkt *Datei|Neu|Website...* Geben Sie den Verzeichnisnamen "ZinsRechner" an und verwenden Sie zunächst als Speicherort das Dateisystem.

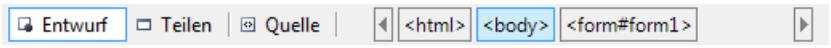
Fügen Sie nun noch dem Projekt eine erste Seite *Default.aspx* hinzu. Der Projektmappenexplorer sollte jetzt folgenden Anblick bieten:



Für uns ist zunächst nur die Datei *Default.aspx* von Interesse. Klicken Sie doppelt darauf, öffnet sich statt des Formulardesigners zunächst die ASP-Quellcodeansicht.

Neben dieser Ansicht können Sie auch die reine Entwurfsansicht oder eine Mischform (Splitview) wählen, bei der Sie sowohl über das Code- als auch das Entwurfs-Fenster verfügen können.

Zwischen den Ansichten können Sie über die drei Schalter am unteren Rand des Designers jederzeit umschalten:



Bei einem Blick in den Werkzeugkasten dürften Sie zunächst viele Gemeinsamkeiten mit den schon bekannten Windows Forms-Controls feststellen.

HINWEIS: Auch wenn es nahe liegend erscheinen mag, für Web Form-Projekte sollten Sie die Controls auf der Seite HTML nur dann verwenden, wenn Sie genau wissen, was Sie da tun. Im Normalfall verwenden Sie bitte die Server-Controls aus den anderen Rubriken, nur diese verfügen über die schon bekannten Eigenschaften und ein entsprechendes Ereignismodell.

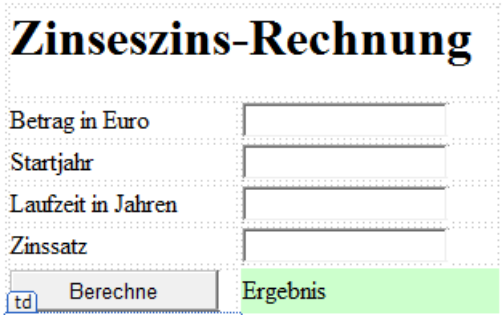
Noch eine Bemerkung auf den Weg:

HINWEIS: Auch wenn einige der Controls noch so komplex und aufwändig in der Gestaltung sind (z.B. *TreeView*), auf dem Client gibt es nur Text, Textboxen, Listboxen, Checkboxen, HTML-Tabellen, Grafiken und einfache Buttons. Alle anderen (Server-) Controls werden automatisch so umgeformt (gerendert), dass sie mit den genannten Controls dargestellt werden können.

39.4.2 Oberflächengestaltung

Der eigentliche Entwurf der Oberfläche entspricht fast vollständig dem Entwurf einer Windows Forms-Oberfläche, ziehen Sie einfach die gewünschten Komponenten in den Designer bzw. in die *Page*.

Platzieren Sie zunächst einige *TextBoxen*, *Labels* und einen *Button* entsprechend folgender Abbildung auf dem Web Form:



Um optische Feinheiten brauchen Sie sich noch nicht zu kümmern, das kann auch in einer späteren Phase erfolgen.

HINWEIS: Wichtig ist jedoch die Verwendung einer HTML-Tabelle für die Positionierung, da Sie andernfalls beim eingestellten Flow-Layout Schwierigkeiten haben, die Controls sinnvoll auszurichten.

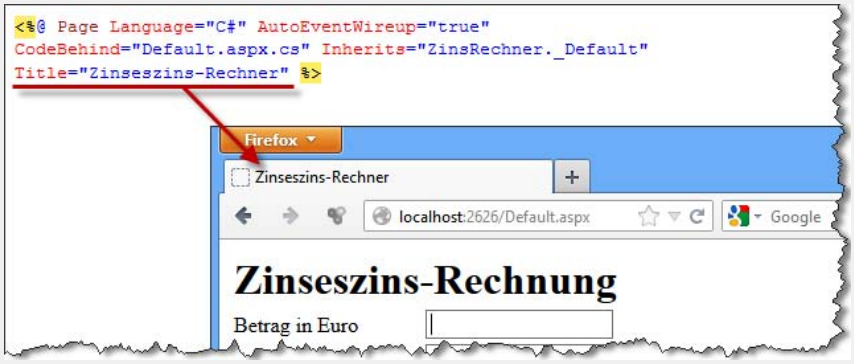
Damit wäre der Oberflächenentwurf vorläufig abgeschlossen, wir können einen ersten Test wagen. Starten Sie einfach mit *F5* und warten Sie ab, was der Explorer Ihnen präsentiert (die Frage nach dem Erstellen einer *Web.Config* bestätigen Sie bitte).

Klicken Sie ruhig einmal auf den Button "Berechne". Auf den ersten Blick passiert zwar nichts, wer jedoch aufmerksam die Statusleiste im Auge behält wird feststellen, dass die Seite erneut vom Server abgerufen wird. Genau dieses Verhalten werden wir im weiteren Verlauf für die Ereignisprogrammierung brauchen.

Formular-Titel setzen

Betrachten Sie obige Abbildung, wird Ihnen sicher der fehlende Titel (Kopfzeile Browser) auffallen. Diesen können Sie über die *DOCUMENT*-Eigenschaften (*Title*) setzen.

BEISPIEL 39.6: Einstellen des Formular-Titels



39.4.3 Ereignisprogrammierung

Im nächsten Schritt wollen wir unserer Anwendung Leben einhauchen, d.h., die eigentliche Berechnung vornehmen. Wer in der Schule schlecht aufgepasst hat bzw. beim wem es zu lange her ist, kann die Zinseszins-Formel aus der Fußzeile[1] entnehmen, alle anderen klicken doppelt auf den Button und tragen die Berechnungsfunktion in den bereits automatisch erzeugten Ereignismethodenrumpf ein ein:

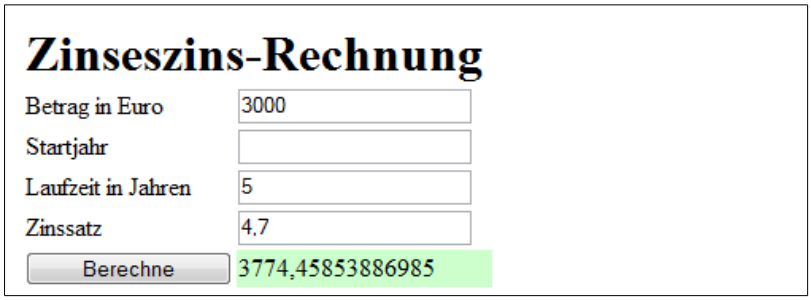
```
protected void Button1_Click(object sender, EventArgs e)
{
    Label1.Text = Convert.ToString(Convert.ToSingle(TextBox1.Text) *
```

[1] Endwert = Anfangswert * (1+Zins/100)^{Laufzeit}

```
                Math.Pow(1 + Convert.ToSingle(TextBox4.Text) / 100,
                Convert.ToSingle(TextBox3.Text)));
}
```

HINWEIS: Auf eine Fehlerbehandlung etc. verzichten wir zunächst.

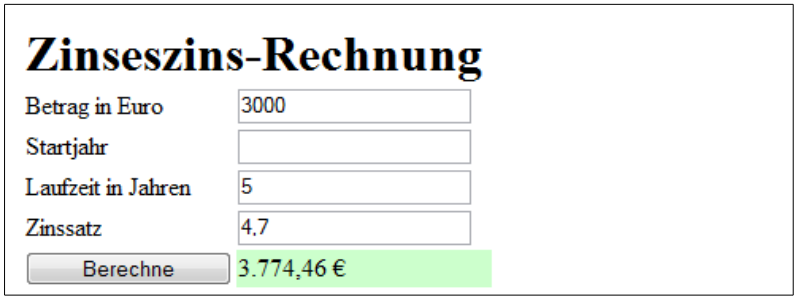
Starten Sie das Programm erneut und füllen Sie die Felder *Betrag*, *Laufzeit* und *Zins* aus. Das Startjahr heben wir uns noch für eine spätere Erweiterung auf.



Das sieht ja schon ganz gut aus, zumal sich der Programmieraufwand bislang in Grenzen hielt. Doch wie sieht es mit einer sinnvollen Endwert-Formatierung aus? Die praktische *ToString*-Methode hilft uns weiter:

```
        Label1.Text = (Convert.ToSingle(TextBox1.Text) *
                    Math.Pow(1 + Convert.ToSingle(TextBox4.Text) / 100,
                    Convert.ToSingle(TextBox3.Text))).ToString("C");
```

Das Ergebnis dürfte zunächst einmal unseren Erwartungen entsprechen:



HINWEIS: Da die Verarbeitung auf dem Server abläuft, gelten dessen Währungseinstellungen, nicht die des Clients!

Bisher sind wir bei unserem Beispiel von vorbildlichen Nutzern ausgegangen, die sofort wissen, dass ein Komma einzugeben ist, und die auch realistische Werte vorgeben. Doch was passiert, wenn nicht?

39.4.4 Ein Fehler, was nun?

Geben Sie statt einer Zahl ein paar Buchstaben ein. Das Resultat dürfte auch bei Ihnen folgende recht ausführliche Webseite sein (siehe folgende Abbildung).

Serverfehler in der Anwendung /.

Die Eingabezeichenfolge hat das falsche Format.

Beschreibung: Unbehandelte Ausnahme beim Ausführen der aktuellen Webanforderung. Überprüfen Sie die Stapelüberwachung, um weitere Informationen über diesen Fehler anzuzeigen und festzustellen, wo der Fehler im Code verursacht wurde.

Ausnahmedetails: System.FormatException: Die Eingabezeichenfolge hat das falsche Format.

Quellfehler:

```
Zeile 23:         protected void Button1_Click(object sender, EventArgs e)
Zeile 24:         {
Zeile 25:             Label1.Text = (Convert.ToSingle(TextBox1.Text) *
Zeile 26:                           Math.Pow(1 + Convert.ToSingle(TextBox4.Text) / 100,
Zeile 27:                           Convert.ToSingle(TextBox3.Text))).ToString("C");
```

Wie Sie sehen, wird Ihnen auch gleich der Quellcode angezeigt, in dem der Fehler aufgetreten ist. Doch keine Panik, eine derart ausführliche und sicher auch nicht immer erwünschte Fehlermeldung erhalten Sie nur auf Ihrem eigenen PC. Der "gemeine" Web-User muss sich mit folgender Meldung zufrieden geben:

Serverfehler in der Anwendung /ZinsRechner.

Laufzeitfehler

Beschreibung: Anwendungsfehler auf dem Server. Aufgrund der aktuellen benutzerdefinierten Fehlereinstellungen für diese Anwendung können die Details des Anwendungsfehlers nicht angezeigt werden.

Details: Sie können die Details dieser Fehlermeldung auf dem lokalen Computer anzeigen, indem Sie ein <customErrors>-Tag in der Konfigurationsdatei web.config erstellen, die sich im Stammverzeichnis der aktuellen Webanwendung befindet. Das mode-Attribut dieses <customErrors>-Tags sollte auf RemoteOnly festgelegt werden. Sie können die Details auf Remotecomputern anzeigen, indem Sie "mode" auf "Off" festlegen.

```
<!-- Web.Config Configuration File -->

<configuration>
    <system.web>
        <customErrors mode="RemoteOnly"/>
    </system.web>
</configuration>
```

Was es damit genau auf sich hat und wie Sie mit Fehlern in ASP.NET-Anwendungen umgehen, zeigt Ihnen Kapitel 42 im Detail.

Einfache Fehlerbehandlung

Wir beschränken uns an dieser Stelle zunächst auf die erste Maßnahme, eine einfache Fehlerbehandlung mit *try catch*.

Die Umsetzung:

```
protected void Button1_Click(object sender, EventArgs e)
{
    try
    {
        Label1.Text = (Convert.ToSingle(TextBox1.Text) *
                    Math.Pow(1 + Convert.ToSingle(TextBox4.Text) / 100,
                    Convert.ToSingle(TextBox3.Text))).ToString("C");
    }
    catch
    {
        Label1.Text = "Error";
    }
}
```

Gibt der Nutzer falsche bzw. unzulässige Werte ein, wird statt des Ergebnisses ein entsprechender Text angezeigt.

39.4.5 Ereignisse von Textboxen

Warum muss der Anwender unbedingt den Button anklicken? Genügt es nicht auch, dass die Neuberechnung automatisch bei der Eingabe vorgenommen wird? Kein Problem! Klicken Sie einfach doppelt auf eine der Textboxen und Sie erhalten den Rahmencode einer *TextChanged*-Ereignis-Methode:

```
protected void TextBox1_TextChanged(object sender, EventArgs e)
{
}
```

Die Betonung liegt auf "Changed", d h., das Ereignis tritt erst **nach** der Änderung auf. Doch wann ist dies der Fall? Mit den Default-Einstellungen wartet der Browser so lange, bis Sie die Eingabe in der *TextBox* mit der Enter-Taste abschließen. Erst dann wird die Meldung an den Server gesendet, die entsprechende Ereignisroutine ausgeführt und die Antwortseite an den Client zurückgeschickt.

So weit, so gut, wenn Sie aber mit der Tabulatortaste zwischen den Textboxen wechseln, passiert nichts. Nutzen Sie die Eigenschaft *AutoPostBack,* um bei jeder Änderung[1] eine Meldung zum Server zu senden bzw. um das Ergebnis zu berechnen.

[1] Eigentlich auch nur beim Fokus-Verlust bzw. beim Verwenden der Enter-Taste.

39.4.6 Ein gemeinsamer Ereignis-Handler

Setzen Sie für alle Eingabefelder die Eigenschaft *AutoPostBack* auf *True* und erzeugen Sie zunächst folgende Ereignismethode:

```
    protected void TextBox1_TextChanged(object sender, EventArgs e)
    {
        try
        {
            Label1.Text = (Convert.ToSingle(TextBox1.Text) *
                          Math.Pow(1 + Convert.ToSingle(TextBox4.Text) / 100,
                          Convert.ToSingle(TextBox3.Text))).ToString("C");
        }
        catch
        {
            Label1.Text = "Error";
        }
    }
```

Diesen Ereignis-Handler weisen Sie auch den weiteren Textboxen zu (Eigenschaften-Editor, Ereignisse). Nach dem Start können Sie sich von der Funktionsfähigkeit überzeugen.

39.4.7 Eingabefokus setzen

Doch ein kleines Manko bei obiger Vorgehensweise bleibt: Mit jedem Seiten-Refresh ist auch der Eingabefokus wieder verändert. Günstiger wäre es, den Fokus je nach *TextBox* um eine Position weiter zu setzen. Nutzen Sie dazu die *Focus*-Methode.

Erweitern Sie also den obigen Ereignishandler um folgende Anweisungen:

```
...
            Label1.Text = "Error";
        }
        if (sender == TextBox1) TextBox2.Focus();
        if (sender == TextBox2) TextBox3.Focus();
        if (sender == TextBox3) TextBox4.Focus();
        if (sender == TextBox4) TextBox1.Focus();
    }
```

Zusätzlich ist es auch möglich, über das *Form*-Element den Anfangsfokus zu setzen. Wechseln Sie dazu in die Quellcode-Ansicht (HTML) und erweitern Sie das *Form*-Element um das Attribut *defaultfocus*:

```
...
  <body>
    <form id="form1" runat="server" defaultfocus= "TextBox1">
...
```

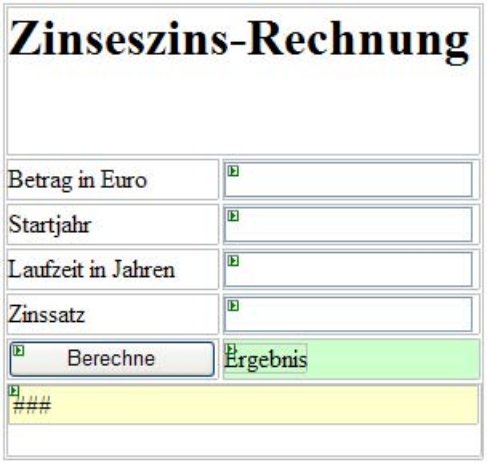
Ein Test dürfte den Kursor zunächst in *TextBox1* auftauchen lassen, jedes *Enter* löst die obige Ereignismethode aus und verschiebt den Fokus auf die nächstfolgende Textbox.

39.4.8 Ausgaben in einer Tabelle

Der etwas nutzlos gewordene Button soll umfunktioniert werden. Statt lediglich eine Endsumme anzuzeigen, sollen zusätzlich auch die Zinsen bzw. die aufgelaufene Summe pro Jahr berechnet werden. Eine Tabelle dürfte da sicher das Mittel der Wahl sein.

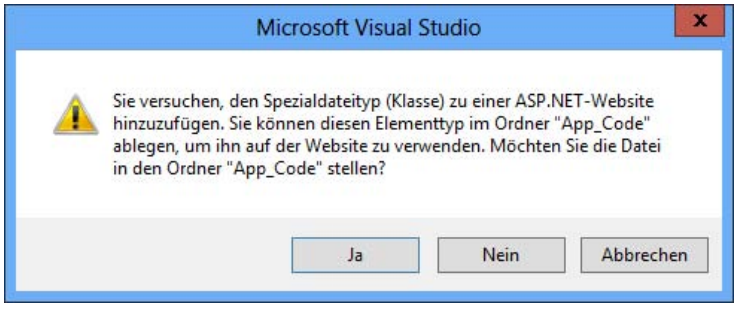
Platzieren Sie also eine *Table*-Komponente (Rubrik *Standard*) auf dem Web Form. Im Designer bestimmen Sie lediglich die Breite der Tabelle (*Width=100%*), alle anderen Formatierungen, die Zeilen- und Spaltenzahl und natürlich auch den Inhalt, legen wir erst zur Laufzeit fest.

HINWEIS: Wesentlich mehr Informationen zur *Table* finden Sie ab Seite 1927.



Die bisherige Ereignisprozedur des Buttons können Sie ruhig wieder löschen, wir erstellen eine neue. Doch bevor es so weit ist, schreiben wir eine neue Klasse *ZinsRechnung*, mit einer statischen Methode *ZinsesZins*, da wir die Berechnung ja nun für mehrere Jahre durchführen müssen.

Wählen Sie dazu den Menüpunkt *Website|NeuesElement hinzufügen* und suchen Sie den Eintrag *Klasse*. Vergeben Sie den Namen *ZinsRechnung.cs*. Nach einem Klick auf OK dürften Sie mit der folgenden Dialogbox konfrontiert werden, vorausgesetzt, Sie arbeiten mit einem ASP.NET-Website-Projekt:



Da nicht mehr alle Web Formulare in einer Assembly zusammengefasst werden, besteht auch kein direkter Zusammenhang zwischen diesen. Möchten Sie formularübergreifend Klassen definieren, müssen diese im Verzeichnis *App_Code* abgelegt werden. Beantworten Sie die obige Frage also mit "Ja".

HINWEIS: Verwenden Sie eine ASP.NET-Anwendung, wird diese in **eine** Assembly kompiliert, wodurch diese Problematik nicht auftritt.

Die neue Klasse:

```
using System;
public static class ZinsRechnung
{
    public static Single ZinsesZins(Single startwert, Single zins, Single laufzeit)
    {
       return (Single)(startwert * Math.Pow(1 + zins / 100, laufzeit));
    }
}
```

Die Ereignisprozedur des *Button*s (auf eine Fehlerbehandlung verzichten wir an dieser Stelle):

```
    protected void Button1_Click(object sender, EventArgs e)
    {
        TableRow r;
        TableCell c;
        int jahr, laufzeit;
        Single startwert, zins;
```

Die Variablen initialisieren:

```
        startwert = Convert.ToSingle(TextBox1.Text);
        jahr = Convert.ToInt32(TextBox2.Text);
        laufzeit = Convert.ToInt32(TextBox3.Text);
        zins = Convert.ToSingle(TextBox4.Text);
```

Wir erzeugen die Kopfzeile der Tabelle:

```
        r = new TableRow();       // neue Zeile erzeugen
        c = new TableCell();      // neue Spalte bzw. Zelle erzeugen
        c.Text = "Jahr";          // Inhalt eintragen
        r.Cells.Add(c);            // Zelle in die Zeile einfügen
        c = new TableCell();
        c.Text = "Wert in Euro";
        r.Cells.Add(c);
        r.BackColor = System.Drawing.Color.LightGray;
        Table1.Rows.Add(r);        // Zeile in die Tabelle einfügen
```

Für alle Jahre berechnen wir die Werte und erzeugen die Tabellenzeilen:

```
        for (int j = 0; j <= laufzeit; j++)
        {
```

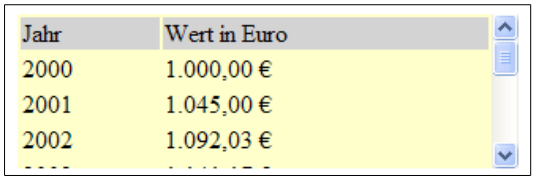
```
            r = new TableRow();
            c = new TableCell();
            c.Text = jahr.ToString();
            r.Cells.Add(c);
            c = new TableCell();
            c.Text = ZinsRechnung.ZinsesZins(startwert, zins, j).ToString("C");
            r.Cells.Add(c);
            jahr++;
            Table1.Rows.Add(r);
        }
    }
```

Damit können wir zur Tat schreiten, einem Test steht nichts mehr im Wege:

39.4.9 Scrollen der Anzeige

Gefällt Ihnen die unter Umständen doch recht lange Tabelle nicht (das hängt von der Anzahl der Jahre ab), können Sie mit einem kleinen Trick für mehr Übersicht sorgen.

Fügen Sie einfach das *Table*-Steuerelement in ein *Panel* ein und legen Sie dessen *Scrollbars*-Eigenschaft auf *Vertical* fest. Jetzt noch die Breite der *Table* mit *300px* angegeben und ein neuer Test kann beginnen:



39.4.10 Zusammenspiel mehrerer Formulare

Vielleicht ist Ihnen das obige Formular auch etwas zu unübersichtlich geworden. Warum kann die jährliche Aufstellung nicht einfach in einem weiteren Web-Form angezeigt werden? Nichts leichter als das.

Erstellen Sie einfach ein neues Web-Form und geben Sie diesem den Namen *Details.aspx*. Fügen Sie ein *Table*-Steuerelement in das neue Web-Form ein.

Für den *Button* in *Default.aspx* selbst erzeugen Sie eine neue Ereignisroutine:

```
protected void Button2_Click(object sender, EventArgs e)
{
    Session["Startwert"] = Convert.ToSingle(TextBox1.Text);
    Session["Jahr"] = Convert.ToInt32(TextBox2.Text);
    Session["Laufzeit"] = Convert.ToInt32(TextBox3.Text);
    Session["Zins"] = Convert.ToSingle(TextBox4.Text);
    Response.Redirect("Details.aspx");
}
```

Globale Variablen?

Was geht hier vor? Über das *Session*-Objekt erzeugen wir quasi globale Variablen, in denen wir die Eingabewerte des Eingabeformulars speichern. Das Detailformular kann diese Werte einfach auslesen und für die Berechnung verwenden. Mit *Response.Redirect("Details.aspx")* wird zum Schluss die zweite Seite aufgerufen.

HINWEIS: Mehr zu diesem Thema finden Sie ab Seite 2043.

Das Page_Load-Ereignis

Doch wo führen wir die Berechnungen im Detailformular aus? Wie bei den Windows Forms auch, können Sie mit dem Laden der Seite (*Page_Load*) eine Ereignisroutine ausführen. Wohlgemerkt passiert dies, **bevor** die Seite vom Server an den Client übertragen wird. Am Client kommt im Weiteren eine fertige HTML-Seite an.

Die Umsetzung:

```
protected void Page_Load(object sender, EventArgs e)
{
    int jahr, laufzeit;  Single startwert, zins;
```

Übernahme der Werte aus den Session-Variablen:

```
        startwert = (Single) Session["Startwert"];
        zins      = (Single) Session["Zins"];
        laufzeit  = (int) Session["Laufzeit"];
        jahr      = (int) Session["Jahr"];
        TableRow  r = new TableRow();       // neue Zeile erzeugen
        TableCell  c = new TableCell();      // neue Spalte bzw. Zelle erzeugen
        c.Text = "Jahr";          // Inhalt eintragen
        r.Cells.Add(c);            // Zelle in die Zeile einfügen
        c = new TableCell();
        c.Text = "Wert in Euro";
        r.Cells.Add(c);
        r.BackColor = System.Drawing.Color.LightGray;
        Table1.Rows.Add(r);        // Zeile in die Tabelle einfügen
        for (int j = 0; j <= laufzeit; j++)
        {
            r = new TableRow();
            c = new TableCell();
            c.Text = jahr.ToString();
            r.Cells.Add(c);
            c = new TableCell();
            c.Text = ZinsRechnung.ZinsesZins(startwert, zins, j).ToString("C");
            r.Cells.Add(c);
            jahr++;
            Table1.Rows.Add(r);
        }
    }
```

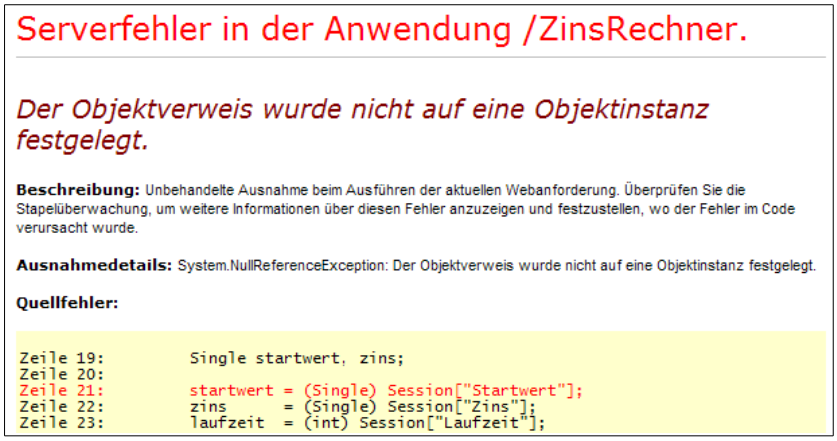
Starten Sie nun die Webanwendung, geben Sie Werte in die Textboxen ein und klicken Sie auf den Button. Es sollte folgendes Formular zu sehen sein:

Jahr	Wert in Euro
2000	1.000,00 €
2001	1.050,00 €
2002	1.102,50 €

39.4.11 Umleiten bei Direktaufruf

Möglicherweise hat Ihre Euphorie von folgender Meldung einen Dämpfer bekommen (siehe folgende Abbildung).

Vermutlich haben Sie aus dem Entwurf heraus gleich *Details.aspx* aufgerufen, so wurden die Variablen nicht initialisiert.



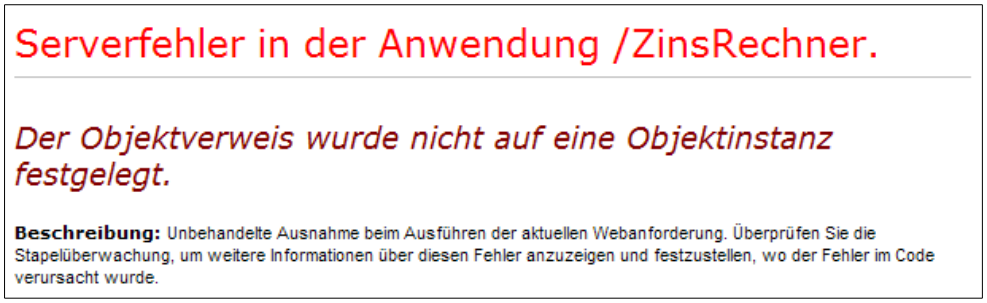
Sie können diesen Fehler ausmerzen, wenn Sie die *Form_Load*-Ereignisprozedur etwas erweitern:

```
...
        int jahr=0, laufzeit=0;
        Single startwert=0, zins=0;
...
        try
        {
            startwert = (Single)Session["Startwert"];
            zins = (Single)Session["Zins"];
            laufzeit = (int)Session["Laufzeit"];
            jahr = (int)Session["Jahr"];
        }
        catch
        {
            Server.Transfer("Default.aspx");
        }
...
```

HINWEIS: Befinden sich undefinierte Einträge in den *Session*-Variablen, springt ASP.NET automatisch auf *Default.aspx*, der Nutzer bekommt *Details.aspx* nie zu sehen.

39.4.12 Ärger mit den Cookies

Sollten Sie, was gar nicht so abwegig ist, mit folgender Fehlermeldung konfrontiert werden, haben Sie entweder einen Tippfehler in Ihrem Beispiel oder Ihr Browser/Firewall blockt Cookies ab:



Die Ursache: Eine Session erfordert einen eindeutigen Bezeichner auf dem Client, der im Standardfall per Cookie realisiert wird. Ist das Erzeugen von Cookies auf dem Client nicht möglich (Ursache siehe oben), können im zweiten Formular nicht mehr die Daten aus dem *Session*-Objekt ausgelesen werden, für den Server fehlt einfach der Zusammenhang zwischen den beiden Formularen.

Für diesen Fall haben sich die MS-Programmierer einen Trick einfallen lassen. Statt lokaler Cookies wird ein virtuelles Verzeichnis (die Session-ID) in den URL eingeschoben, dessen Name bei relativ miteinander verlinkten Seiten durch den Browser automatisch immer wieder eingefügt wird.

Bevor dieser Mechanismus greift, müssen wir jedoch in der *Web.config* eine Änderung vornehmen. Fügen Sie bitte folgenden Abschnitt ein:

```
<configuration>
  <appSettings/>
    <system.web>
     <compilation debug="true" targetFramework="4.5"/>
     <httpRuntime targetFramework="4.5"/>
     <sessionState mode="InProc" cookieless="true" timeout="20"/>
```

Wichtig ist der Wert des Attributs *cookieless*. Speichern Sie die Datei *Web.config* ab, um die Änderungen für den Server zu übernehmen.

Bei einem erneuten Start Ihrer Anwendung sollte jetzt die Adresszeile für *Default.aspx* wie folgt aussehen:

Beachten Sie die in Klammern gespeicherte Session-ID. Beim Aufruf von *Details.aspx* ist der Browser dafür verantwortlich, die Adresse entsprechend zu erweitern. Dies erfolgt automatisch, da wir nur eine relative Pfadangabe für *Details.aspx* angegeben hatten:

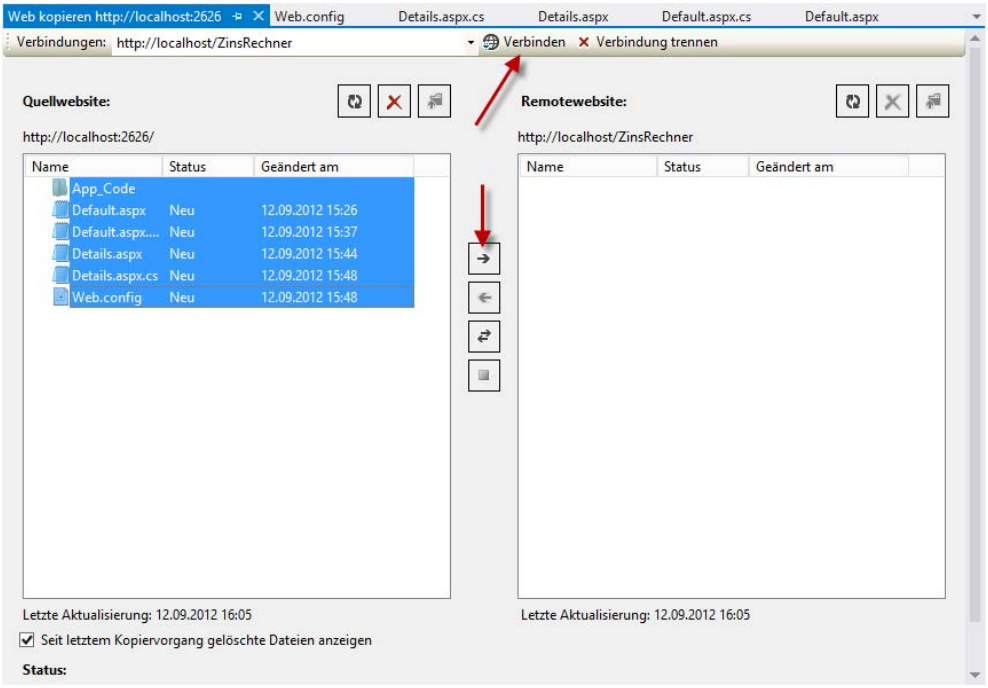
Damit wäre auch das kleine Problem mit den Cookies bzw. der Session-ID aus der Welt geschafft.

39.4.13 Export auf den IIS

Nachdem unsere Anwendung nun schon ganz gut funktioniert, wollen wir den letzten Schritt wagen, die Veröffentlichung auf einem richtigen Web-Server.

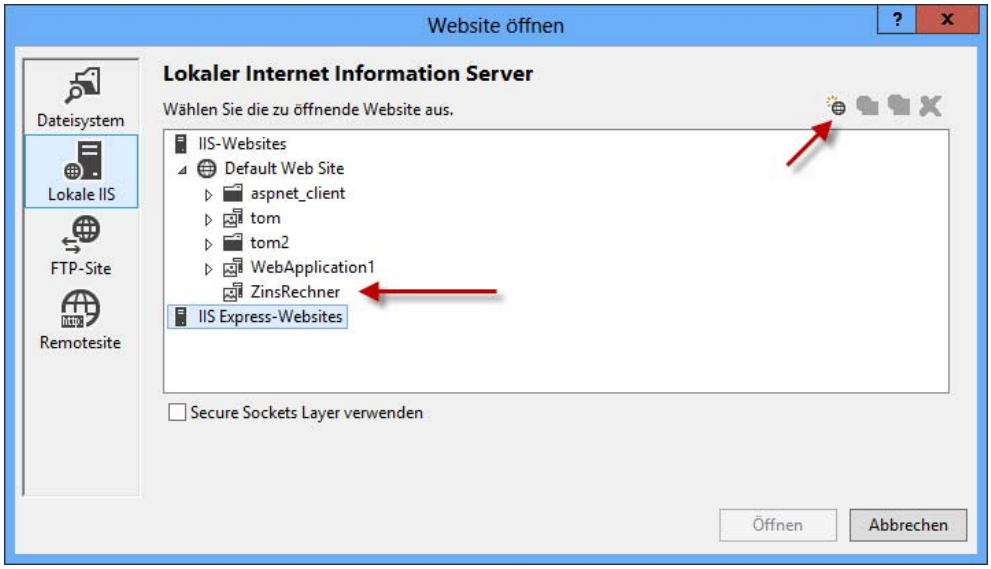
Zur Erinnerung: Bisher haben wir nur mit dem Test-Server von Visual Studio oder dem IIS-Express gearbeitet, der schon von einem zweiten PC aus nicht mehr erreichbar ist.

Für den Export unserer Web-Anwendung bietet Visual Studio ein eigenes Tool, das Sie über den *Menüpunkt Website|WebSite kopieren* erreichen[1]:



Klicken Sie im Toolbar den Button *Verbinden* und wählen Sie die Zugangsart zu Ihrem Webserver. Wir belassen es beim Export unseres Webs bei einem lokal laufenden IIS und erzeugen dort eine neue Webanwendung:

[1] Vorausgesetzt, Sie haben ein ASP NET-Website-Projekt erstellt.



Nach dem Kopieren können Sie den Internet Explorer aufrufen und den Zugriff auf die Webanwendung testen.

Unser kurzer Rundflug über einige wichtige ASP.NET-Features wäre damit beendet. Mehr als einen ersten Eindruck konnten Sie allerdings nicht gewinnen, und wir kommen nicht umhin, Sie in den folgenden Kapiteln mit zahlreichen weiteren Details zu foltern.

39.5 Tipps & Tricks

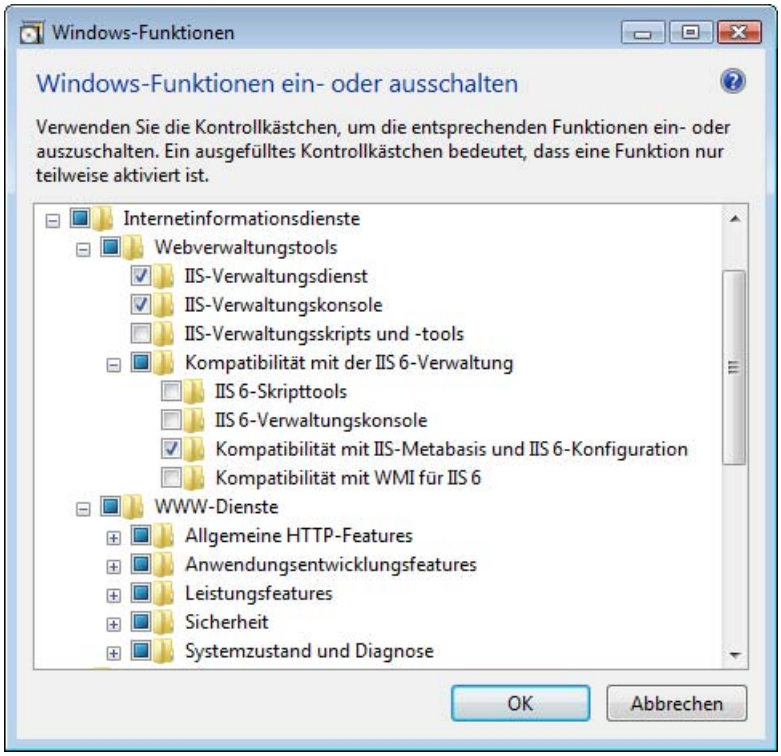
39.5.1 Nachinstallieren IIS 7 bzw. 7.5 (Windows 7)

Nach der Installation von Visual Studio 2012 möchten Sie Ihre ASP.NET-Projekte sicher auch einmal auf Ihrem lokalen IIS laufen lassen. Doch irgendwas funktioniert nicht richtig, Visual Studio verweigert die Zusammenarbeit und irgendwann kommt die Erkenntnis, dass der IIS (*Internet Information Server*) noch gar nicht installiert ist.

HINWEIS: Unter Windows Vista/Windows 7/8 wird der IIS standardmäßig **nicht** installiert!

Nachinstallieren

Wählen Sie in der Systemsteuerung (*Programme und Features*) auf der linken Seite *Windows Funktionen ein- oder ausschalten* und aktivieren Sie die WWW-Dienste sowie die unten gezeigten Verwaltungsdienste:



HINWEIS: Achten Sie auch auf die Aktivierung der Option "Kompatibilität mit IIS-Metabasis und IIS 6-Konfiguration".

ASP.NET registrieren

Nach erfolgter Installation müssen Sie noch ASP.NET registrieren. Wechseln Sie dazu in die Windows-Kommandozeile (*Start|Alle Programme|Zubehör|Eingabeaufforderung*) und geben Sie folgende Anweisung für das Framework 2.x ein:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\aspnet_regiis.exe -i
```

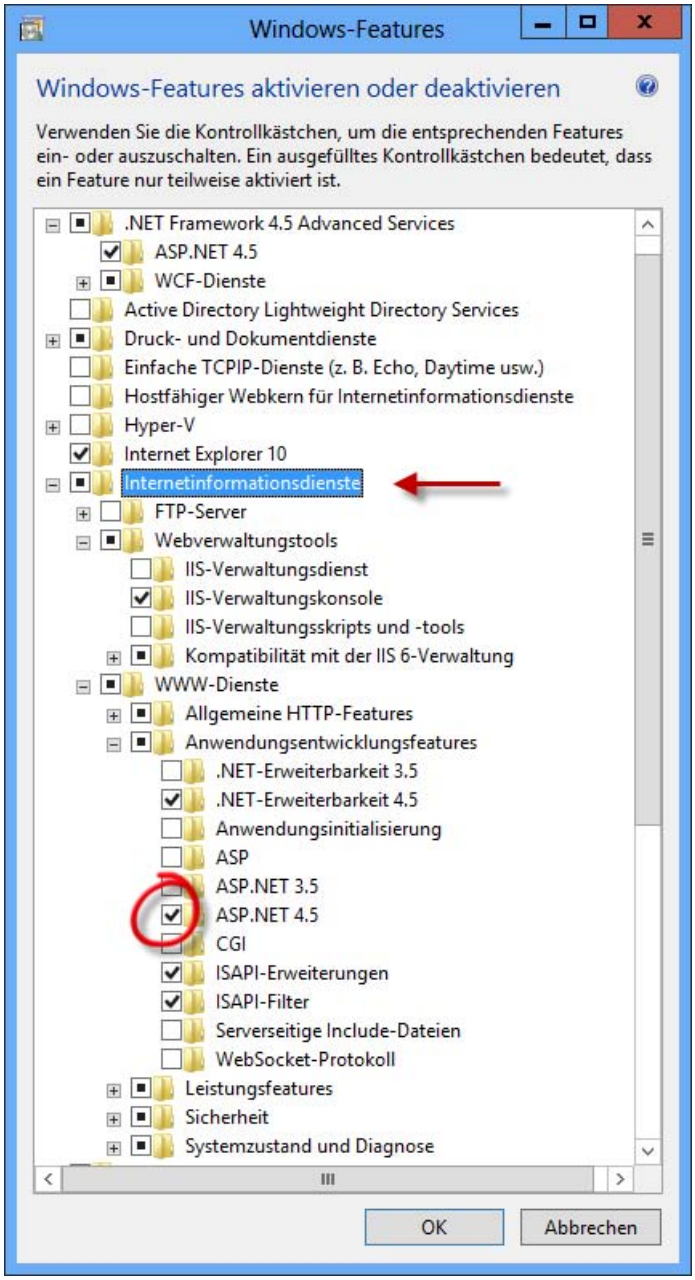
bzw. für .NET 4:

```
C:\WINDOWS\Microsoft.NET\Framework\v4.0.11208\aspnet_regiis.exe -i
```

Abschließend können Sie in Visual Studio auch auf dem lokalen IIS neue Projekte anlegen bzw. vorhandene Webprojekte auf den IIS exportieren.

39.5.2 Nachinstallieren IIS8 (Windows 8)

Auch hier öffnen Sie zunächst, wie im vorhergehenden Abschnitt, die Systemsteuerung (Programme und Features) und klicken auf den Link *Windows Funktionen ein- oder ausschalten*, um die WWW-Dienste sowie die unten gezeigten Verwaltungsdienste zu aktivieren.



Achten Sie in diesem Zusammenhang auf den ASP.NET 4.5-Eintrag, der in obiger Abbildung hervorgehoben sind. Dieser installiert die ASP.NET-Unterstützung für den IIS, der spätere Aufruf von *aspnet_regiis.exe* ist nicht mehr nötig.

Kapitel 40

Übersicht ASP.NET-Controls

In diesem Kapitel wollen wir die wichtigsten Verwendungsmöglichkeiten der einfachen Web-Steuerelemente aufzeigen.

HINWEIS: Wer an Datenbindung interessiert ist, der darf gleich zum folgenden Kapitel 41 weiterblättern.

40.1 Einfache Steuerelemente im Überblick

Der folgende Abschnitt kann und soll keine umfassende Übersicht aller Steuerelemente und deren Eigenschaften sein. Wir geben Ihnen lediglich einen groben Überblick und einige Anregungen, wie Sie die Steuerelemente sinnvoll einsetzen können.

HINWEIS: In den Buch-Beispielen finden Sie ein Web-Projekt *Steuerelemente*, das in einer bunten Zusammenstellung diverse Einsatzfälle demonstriert.

40.1.1 Label

Wie auch bei den Windows Forms-Steuerelementen dient das *Label* der Darstellung statischer Texte. Im Unterschied zu reinem HTML-Text können Sie jedoch aus dem Code heraus auf diesen Text und seine Eigenschaften (Font, Farbe etc.) zugreifen.

BEISPIEL 40.1: Ändern der Beschriftung und der Textgröße zur Laufzeit

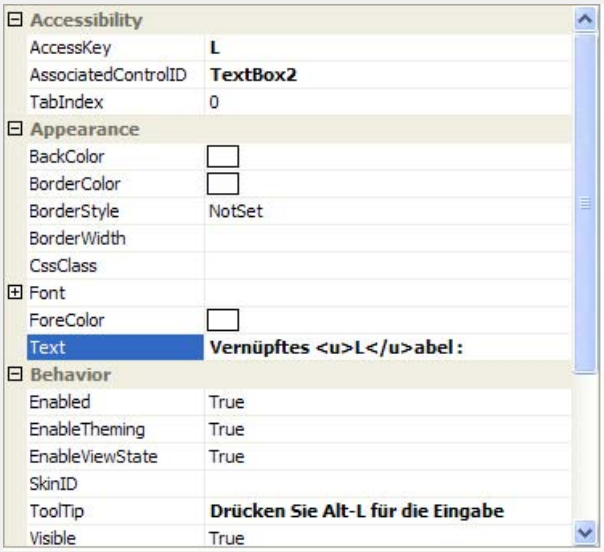
C#
```
protected void Button9_Click(object sender, EventArgs e)
{
  Label1.Text = "Neue Beschriftung";
  Label1.ForeColor = Color.Beige;
}
```

AccessKey/AssociatedControlID

Interessant ist das *Label* auch für das Zuweisen von Hotkeys, d h., mit der Auswahl ALT+<Zugriffstaste> springt der Eingabefokus in das gewünschte Steuerelement. So weit – so gut, werden Sie denken, aber was nützt das bei einem Label in das keine Eingaben vorgenommen werden können? Dies stimmt zwar, Sie haben jedoch die Möglichkeit, über die Eigenschaft *AssociatedControlID* zum Beispiel eine *TextBox* zuzuordnen.

BEISPIEL 40.2: Mit ALT+L soll in *TextBox2* gewechselt werden. Im *Label* soll der betreffende Buchstabe unterstrichen werden.

Setzen Sie dazu die Eigenschaften *AccessKey*, *AssociatedControlID*, *Text* und *ToolTip* wie in der folgenden Abbildung gezeigt:



Datenbindung

An dieser Stelle ein kleiner Vorgriff, denn wir gehen im Kapitel 41 detailliert auf das Thema ein.

Das *Label*, wie zum Beispiel auch die *TextBox*, lässt sich mit wenig Aufwand an eine Datenquelle binden. Voraussetzung ist allerdings, dass sich das Label in einer gebundenen *FormView* befindet.

BEISPIEL 40.3: Bindung an das Feld *Description*

HTML

```
<asp:FormView ID="FormView1" runat="server"
              DataSourceID="AccessDataSource2" DataMember="DefaultView">
    <ItemTemplate>
        <asp:Label ID="Label4" runat="server"
                   Text='<%# Eval("Description") %>'></asp:Label>
    </ItemTemplate>
</asp:FormView>
```

Alternativ können Sie das *Label* zum Beispiel auch an eine Variable binden.

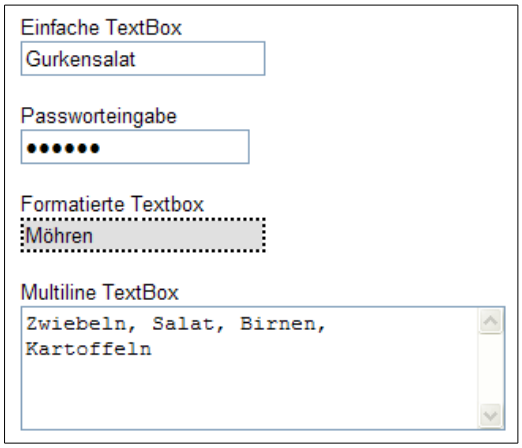
BEISPIEL 40.4: Variablendefinition und Aufruf der *Bind*-Methode

C#
```
...
 public String MeineVariable = "Hallo User";

 protected void Page_Load(object sender, EventArgs e)
  {
   if (!Page.IsPostBack)
       Label5.DataBind();
   ...
```

HTML
Die *Label*-Eigenschaften:
```
<asp:Label ID="Label5" runat="server" Text="<%# MeineVariable %>"
           Font-Bold="True"></asp:Label>
```

40.1.2 TextBox

Die *TextBox*, das wird Sie sicher nicht überraschen, dient der Texteingabe. Über die Eigenschaft *TextMode* steuern Sie das Aussehen. Möglich ist neben einer einzeiligen (Default), mehrzeiligen (*MultiLine*) oder einer verdeckten Eingabe (*Password*) auch die Verwendung für *Color*, *Date*, *DateTime*, *EMail*, *Number* etc. Letztere Werte entsprechen den *type*-Attributen der HTML 5-Spezifikation und werden nur teilweise durch die einzelnen Browser unterstützt..



Höhe und Breite der *TextBox* bestimmen Sie mit den Eigenschaften *Rows* bzw. *Colums*, die in diesem Fall jedoch die Anzahl der Zeichen bzw. Zeilen bezeichnen.

HINWEIS: Das Ereignis *TextChanged* tritt erst nach einem *PostBack* auf. Dies können Sie durch Setzen der Eigenschaft *AutoPostBack* initiieren.

40.1.3 Button, ImageButton, LinkButton

Über den Sinn und die Funktion von Buttons bräuchten wir uns eigentlich nicht länger auszulassen, wenn es sich nicht um Web-Programmierung handeln würde.

BEISPIEL 40.5: Verschiedene Formatierungsmöglichkeiten für Buttons

Click-Ereignis

Hierzu gibt es wohl nicht viel zu erklären.

BEISPIEL 40.6: Verwendung des *Click*-Events, um eine andere Seite aufzurufen

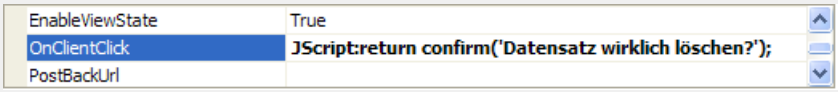
C#
```
protected void Button1_Click(object sender, EventArgs e)
{
  Server.Transfer("default2.aspx");
}
```

OnClientClick-Eigenschaft

Nach der reinen ASP.NET-Lehre soll das *Click*-Ereignis auf dem Server ausgeführt werden. Das ist auch so, doch viele Programmierer möchten schon auf dem Client Fehlerprüfungen vornehmen oder Sicherheitsabfragen (MessageBox) anzeigen. Aus diesem Grund gibt es neben dem oben genannten Ereignis auch eine Eigenschaft *OnClientClick* (ja, es ist eine Eigenschaft!), die ein Client-Script (VBScript/JScript) auswerten kann.

Ist der Rückgabewert *True*, wird ein Postback durchgeführt, andernfalls bleibt es beim Ausführen des Client-Scripts.

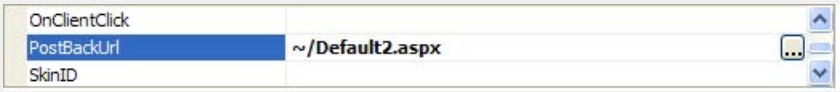
BEISPIEL 40.7: Anzeigen einer Sicherheitsabfrage auf dem Client



PostbackUrl-Eigenschaft

Geht es um das einfache Navigieren zu einer anderen Webseite, können Sie auch von der Eigenschaft *PostBackUrl* Gebrauch machen.

BEISPIEL 40.8: Verwendung *PostBackUrl*



Der Vorteil dieser Vorgehensweise: Per *Cross Page Posting* können Sie in der aufgerufenen Seite auf die Controls der aufrufenden Seite zugreifen.

BEISPIEL 40.9: Zugriff auf die aufrufende Seite

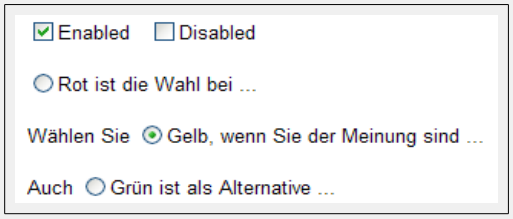
C#

```
protected void Page_Load(object sender, EventArgs e)
{
  if (Page.PreviousPage != null)
  {
    Button btn = (Button) PreviousPage.FindControl("Button3");
    Response.Write(btn.Text + "  in Default.aspx wurde angeklickt!");
  }
}
```

40.1.4 CheckBox, RadioButton

Wem die *CheckBoxList*/*RadioButtonList* zu starr ist bzw. wer nur einen Eintrag anzeigen will, der kann auch *CheckBox*- bzw. *RadioButton*-Controls einzeln auf dem WebForm platzieren.

BEISPIEL 40.10: Verwendung von *CheckBox* und *RadioBox*



Beide Controls dienen der Anzeige von Zuständen (*True*/*False*), mit dem Unterschied, dass bei einem *RadioButton* nur immer ein Control der Gruppe den Zustand *Checked=True* haben kann.

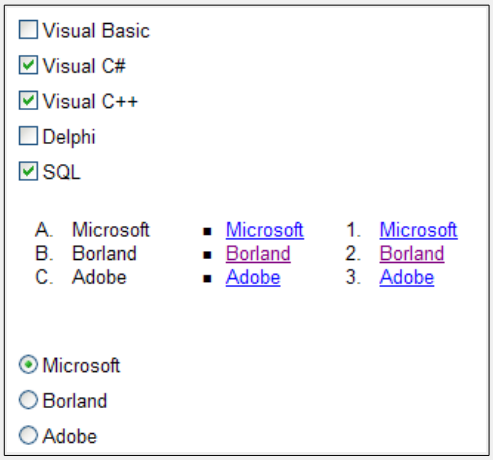
Die *RadioButton*s werden über die Eigenschaft *GroupName* zu einer logischen Gruppe verknüpft. Wo die Controls auf der Seite platziert werden ist unerheblich.

HINWEIS: Soll eine Zustandsänderung auch auf dem Server bemerkt werden, müssen Sie *AutoPostBack* auf *True* setzen.

40.1.5 CheckBoxList, BulletList, RadioButtonList

Die "großen Brüder" von *CheckBox* und *RadioButton* bieten bereits eine komplett formatierte Liste mit fertiger Gruppierung für die Anzeige logischer Zustände.

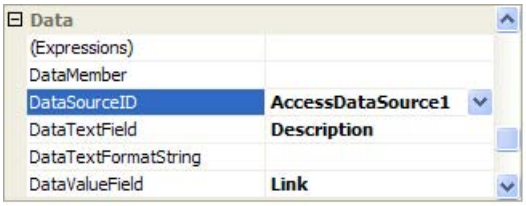
BEISPIEL 40.11: ***CheckBoxList, BulletList* und *RadioButtonList***



HINWEIS: Die obige Abbildung zeigt u.a. einige Formatierungsmöglichkeiten für die *BulletList*. So ist zum Beispiel mittels Datenbindung eine einfache Link-Liste realisierbar, die Zieladresse wird nach einem Klick automatisch angesprungen.

Datenbindung

Ein weiterer wichtiger Unterschied ist die wesentlich vereinfachte Datenbindung ohne die Verwendung einer *FormView*.



Dazu verfügen alle drei genannten Controls über die Eigenschaften *DataMember* (welche Tabelle), *DataSourceId* (Id der Datenquelle), *DataTextField* (die Beschriftung), *DataTextFormatString* (Beschriftung formatieren), *DataValueField* (der zurückgegebene Wert).

HINWEIS: Wichtig ist in diesem Zusammenhang auch die Eigenschaft *AppendDataBoundItems* (*True*/*False*), die es ermöglicht, zu schon statisch vorhandenen Einträgen noch zusätzliche Einträge aus der Datenquelle hinzuzufügen.

BulletList

Auch wenn nur von "Bullet..." die Rede ist, diese Liste kann auch durchnummeriert und mit Buchstaben oder Grafiken gekennzeichnet werden. Nutzen Sie dazu die Eigenschaften *BulletStyle*:

- *BulletStyle.Numbered*
- *BulletStyle.LowerAlpha*
- *BulletStyle.UpperAlpha*
- *BulletStyle.LowerRoman*
- *BulletStyle.UpperRoman*
- *BulletStyle.Disc*
- *BulletStyle.Circle*
- *BulletStyle.Square*
- *BulletStyle.CustomImage*
- und optional *BulletImageUrl* (Adresse einer Grafik).

Auch die Eigenschaft *DisplayMode* verdient einen zweiten Blick, können Sie doch damit automatisch eine funktionierende Hyperlink-Liste oder *LinkButton*s erzeugen lassen (siehe obiges Beispiel).

40.1.6 Table

Hier handelt es sich um eine "einfache" Tabelle, der Sie sowohl zur Entwurfs- als auch zur Laufzeit Zeilen und Spalten hinzufügen können, die Sie nachfolgend mit Inhalten füllen. Die möglichen Formatierungen orientieren sich eng am endgültigen Resultat dieses Server-Controls: eine einfache HTML-Table (Zellabstände, Rahmen, horizontale/vertikale Ausrichtung, Schriftarten etc.).

BEISPIEL 40.12: Ausgabe einer Tabelle mit zwei Spalten

C#

```
for (int zeilen = 0; zeilen < 10; zeilen++)
{
  TableRow row = new TableRow();
  for (int spalten = 0; spalten < 2; spalten++)
  {
    TableCell cell = new TableCell();
    cell.Text = zeilen.ToString() + " : " + spalten.ToString();
    row.Cells.Add(cell);
  }
  Table1.Rows.Add(row);
}
```

BEISPIEL 40.12: Ausgabe einer Tabelle mit zwei Spalten

Ergebnis

Die Tabelle im Browser:

0 : 0	0 : 1
1 : 0	1 : 1
2 : 0	2 : 1
3 : 0	3 : 1
4 : 0	4 : 1
5 : 0	5 : 1
6 : 0	6 : 1
7 : 0	7 : 1
8 : 0	8 : 1
9 : 0	9 : 1

HTML

Der erzeugte HTML-Code:

```
<table id="Table1" cellspacing="5" rules="all" border="2" style="border-width:2px;border-
style:solid;">
 <tr>
   <td>0 : 0</td><td>0 : 1</td>
 </tr><tr>
   <td>1 : 0</td><td>1 : 1</td>
 </tr><tr>
   <td>2 : 0</td><td>2 : 1</td>
...
 </tr>
</table>
```

Ist die Tabelle erst einmal erzeugt, können Sie auch direkt auf einzelne Zellen zugreifen.

BEISPIEL 40.13: Zeile 2, Spalte 1 verändern

C#

```
Table1.Rows[1].Cells[0].Text = "Beschriftung";
```

BEISPIEL 40.14: Zellen formatieren

C#

```
Table1.Rows[1].Cells[0].Font.Bold = true;
Table1.BackColor = Color.BlanchedAlmond;
```

BEISPIEL 40.15: Eine Spalte formatieren

C#

```
foreach (TableRow row in Table1.Rows)
{
  row.Cells[1].BackColor = Color.Gold;      ...
}
```

HINWEIS: Selbstverständlich können Sie auch Steuerelemente in Tabellenzellen unterbringen, nutzen Sie dazu die *cell.Controls.Add*-Methode.

40.1.7 Hyperlink

Die Server-Control-Variante des "gemeinen" HTML-Hyperlinks. Neben den diversen Möglichkeiten zur optischen Formatierung sind eigentlich nur die Eigenschaften *NavigateUrl* (Zieladresse) und *Target* (welches Zielfenster) interessant:

HyperLink

Der erzeugte HTML-Code:

```
<a id="HyperLink1" href="http://www.doko-buch.de" style="background-color:#FFFFC0;border-
color:#E0E0E0;font-size:Larger;">HyperLink</a>
```

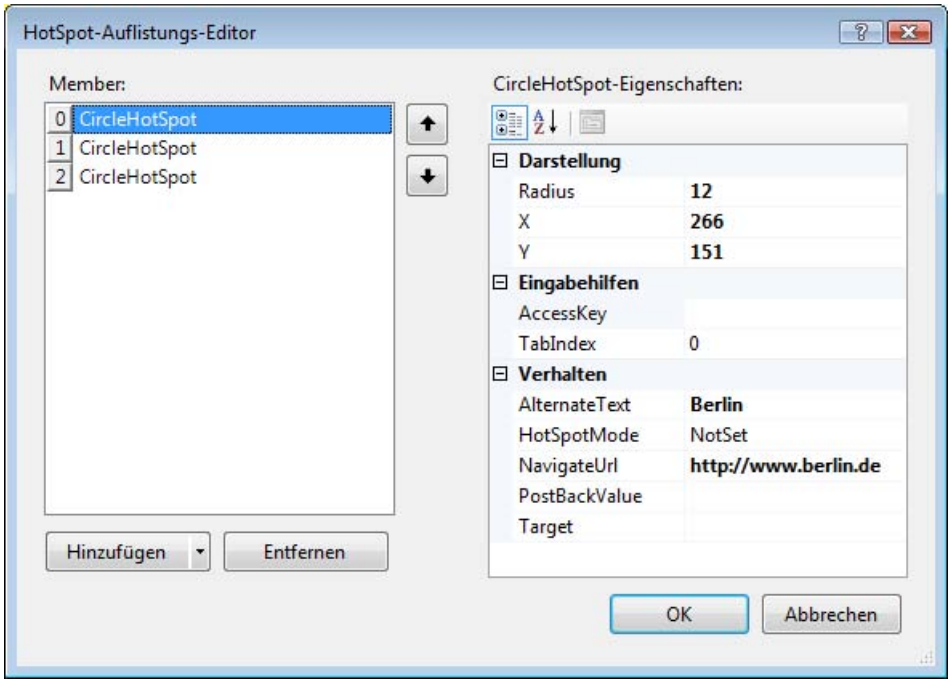
40.1.8 Image, ImageMap

Über die Bedeutung bzw. Verwendung eines *Image*-Controls brauchen wir Sie sicher nicht aufzuklären, die wichtigste Eigenschaft dürfte *ImageUrl* sein, d h. die Adresse der enthaltenen Grafik.

Etwas komplexer und damit auch interessanter ist *ImageMap*, eine Mischung aus *Image*-Control und nicht sichtbaren Hotspots, die sowohl für die direkte Navigation zu einer Webadresse als auch für ein PostBack konfiguriert werden können.

In den Beispieldaten zum Buch finden Sie eine Demo, die im Hintergrund eine Karte der Bundesrepublik Deutschland anzeigt, die Städte Berlin, Hamburg und Bremen sind als kreisförmige Hotspots definiert.

Für die Hotspot-Definition steht Ihnen ein Assistent (Eigenschaft *HotSpots*) zur Verfügung, der jedoch lediglich eine textuelle und keine grafische Unterstützung bietet:



Der HTML-Quellcode für die Beispielanwendung:

```
<asp:ImageMap ID="ImageMap1" runat="server" ImageUrl="~/deutschland.gif"
OnClick="ImageMap1_Click">
 <asp:CircleHotSpot AlternateText="Berlin" NavigateUrl="http://www.berlin.de"
                    Radius="12" X="266" Y="151" />
 <asp:CircleHotSpot AlternateText="Hamburg" NavigateUrl="http://www.hamburg.de"
                    Radius="12" X="152" Y="95" />
 <asp:CircleHotSpot AlternateText="Bremen" HotSpotMode="PostBack"
                    NavigateUrl="http://www.bremen.de" PostBackValue="Bremen"
                    Radius="6" X="110" Y="118" />
```

Der erzeugte HTML-Code auf dem Client:

```
<img id="ImageMap1" src="deutschland.gif" usemap="#ImageMapImageMap1" style="border-
width:0px;" /><map name="ImageMapImageMap1">
 <area shape="circle" coords="266,151,12" href="http://www.berlin.de" title="Berlin"
   alt="Berlin" />
  <area shape="circle" coords="152,95,12" href="http://www.hamburg.de" title="Hamburg"
   alt="Hamburg" />
  <area shape="circle" coords="110,118,6" href="javascript:__doPostBack('ImageMap1','2')"
   title="Bremen" alt="Bremen" />
```

HINWEIS: Sie haben die Wahl zwischen rechteckigen, kreisförmigen und polygonalen Hotspots.

Ist der *HotSpotMode* auf *PostBack* gesetzt, können Sie im *Click*-Ereignis über den Parameter *e.PostBackValue* die jeweilige Hotspotzone ermitteln:

```
protected void ImageMap1_Click(object sender, ImageMapEventArgs e)
{
  if (e.PostBackValue == "Bremen")
    Server.Transfer("deutschland.gif");
}
```

40.1.9 Calendar

Unverzichtbar für die Auswahl von Datumswerten bzw. -bereichen ist das *Calendar*-Control. Neben den diversen Formatierungsmöglichkeiten, die Sie auch durch die Autoformat-Funktion setzen können, ist auch die Eigenschaft *SelectionMode* interessant, die darüber entscheidet, ob Schaltflächen für die Wochen- bzw. Monatsauswahl angezeigt werden (siehe folgende Abbildung):

<	September 2012					>
Mo	Di	Mi	Do	Fr	Sa	So
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

<	September 2012						>
>>	Mo	Di	Mi	Do	Fr	Sa	So
>	27	28	29	30	31	1	2
>	3	4	5	6	7	8	9
>	10	11	12	13	14	15	16
>	17	18	19	20	21	22	23
>	24	25	26	27	28	29	30
>	1	2	3	4	5	6	7

Zur Laufzeit kann der Nutzer dann auch eine Woche oder einen ganzen Monat selektieren:

<	September 2012					>
>>	Mo	Di	Mi	Do	Fr	Sa	So
>	27	28	29	30	31	1	2
>	3	4	5	6	7	8	9
>	10	11	12	13	14	15	16
>	17	18	19	20	21	22	23
>	24	25	26	27	28	29	30
>	1	2	3	4	5	6	7

Woche ab 10.09.2012

BEISPIEL 40.16: Die Auswahl können Sie wie folgt ermitteln:

C#

```
protected void Calendar2_SelectionChanged(object sender, EventArgs e)
{
  switch (Calendar2.SelectedDates.Count)
```

BEISPIEL 40.16: Die Auswahl können Sie wie folgt ermitteln:

```
  {
    case (0):
      Label6.Text = "Keine Auswahl";
      break;
    case (1):
      Label6.Text = "Tag: " + Calendar2.SelectedDate.ToShortDateString();
      break;
    case (7):
      Label6.Text = "Woche ab " + Calendar2.SelectedDate.ToShortDateString();
      break;
    default:    //Month
      Label6.Text = "Monat ab " + Calendar2.SelectedDate.ToShortDateString();
      break;
  }
}
```

40.1.10 Panel

Sicher auch von den Windows Forms-Controls her bekannt, ist das *Panel*, das auf Web Forms die gleichen Aufgaben übernimmt.

Einige kleine Unterschiede gibt es dennoch, wie die folgende Abbildung zeigt:

So kann ein *Panel* einen *GroupingText* (Kopfzeile) aufweisen, es können vertikale und horizontale *Scrollbars* eingefügt werden und der *Wrap*-Mode bestimmt, ob sich die Breite des Panels an den Inhalt oder der Inhalt an das Panel anpasst.

Interessant ist auch die Möglichkeit, mehrere Controls im *Panel* mit einer Anweisung (*Visible= False*) auszublenden.

40.1.11 HiddenField

Ein nicht sichtbares Server-Control, das einen Wert (*Value*) aufweisen kann. So weit – so gut, doch brauchen wir dafür nicht Variablen?

Das stimmt zwar, doch wo existieren diese Variablen? Auf dem Server, und damit kommen wir der Verwendung des Controls schon etwas näher. Nicht jeder Programmierer ist mit den Möglichkeiten der serverseitigen Programmierung vollauf zufrieden. Teilweise lassen sich viele Dinge schon auf dem Client mittels Scripting (z.B. Eingabeprüfung mit JScript) realisieren. Doch wie sollen auf ein-

fache Weise Informationen zwischen Client-Programmierung und Server-Programmierung ausgetauscht werden? Genau hier setzt das *HiddenField*-Steuerlement an.

BEISPIEL 40.17: Auf dem Server können Sie den *Value*-Wert des Controls bequem über das Ereignis *ValueChanged* auswerten (oder zu jedem anderen Zeitpunkt):

C#

```
protected void HiddenField1_ValueChanged(object sender, EventArgs e)
{
   Button7.Text = "Wert wurde durch Client geändert!!";
}
```

JavaScript

Auch auf dem Client können Sie den Wert per JScript auslesen und ändern. Die Funktion wird in unserem Beispiel durch einen Klick auf den *Button7* ausgelöst:

```
...
<asp:HiddenField ID="HiddenField1" runat="server"
                 OnValueChanged="HiddenField1_ValueChanged" Value="222" />
<asp:Button ID="Button7" runat="server" OnClientClick="myFunction();"
            Text="HiddenField ändern" UseSubmitBehavior="False" /></td>
...
</body></html>

<script language="javascript">
  <!--
  function myFunction()
  {
    form1.HiddenField1.value = "abcdefghij";
  }
  -->
</script>
```

40.1.12 Substitution

Das *Substition*-Control spielt im Zusammenhang mit dem Zwischenspeichern von ASP.NET-Seiten (Caching) eine Rolle. Über dieses Control können Sie HTML-Code generieren, der trotz aktivem Caching abgearbeitet wird.

BEISPIEL 40.18: Cache aktivieren und Ausgeben der aktuellen Zeit im normalen ASP.NET-Code und mittels *Substitution*-Control.

C#

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
         Inherits="_Default" %>
<%@ OutputCache Duration="60" VaryByParam="none" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<script runat="server">
    static string mySubstitution(HttpContext context)
    {
```

BEISPIEL 40.18: Cache aktivieren und Ausgeben der aktuellen Zeit im normalen ASP.NET-Code und mittels *Substitution*-Control.

```
        return DateTime.Now.ToString();
    }
</script>
<html ...
...
   Seite erstellt am: <%= DateTime.Now.ToString() %>
   <br />
   Reale Zeit:
   <asp:Substitution ID="Substitution1" runat="server" MethodName="mySubstitution" />
```

Ergebnis

Den Unterschied sehen Sie erst, wenn Sie die Seite mittels Refresh erneut anzeigen.

Vor dem Refresh:

Seite erstellt am:
12.09.2012 20:54:43
Reale Zeit: 12.09.2012 20:54:43

Nach dem Refresh:

Seite erstellt am:
12.09.2012 20:55:05
Reale Zeit: 12.09.2012 20:56:02

40.1.13 XML

Das XML-Control fungiert quasi als Platzhalter für ein neu zu erstellendes HTML-Dokument. Dieses wird zur Laufzeit aus einem XML-File (die eigentlichen Daten) und einem XSL-File (Transformationsanweisungen in XSLT) generiert.

Die XML-Daten

Ein einfaches XML-Dokument (*Daten.xml*) genügt uns:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Adressbuch>
  <Adresse>
    <Name>Erich Mielke</Name>
    <Telefon>01900-123546</Telefon>
  </Adresse>
  <Adresse>
    <Name>Anton Berger</Name>
    <Telefon>01900-312313</Telefon>
  </Adresse>
  <Adresse>
    <Name>Paul Schmidt</Name>
    <Telefon>0800-64734343</Telefon>
```

```
  </Adresse>
  <Adresse>
    <Name>Gerhard Rosengarten</Name>
    <Telefon>03232-23432434</Telefon>
  </Adresse>
</Adressbuch>
```

Die Transformationsdateien

Variante 1 unserer XSL-Transformation (*EinfacheListe.xsl*):

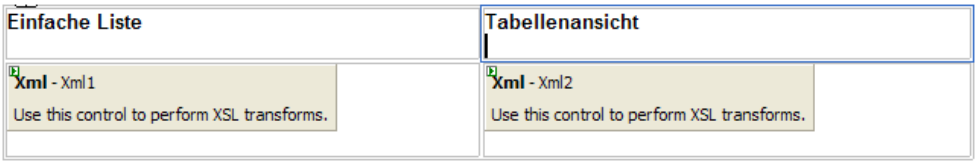
```
<?xml version="1.0" encoding="ISO-8859-1"?><xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
  <html>
  <body>
    <table border="0">
    <xsl:for-each select="Adressbuch/Adresse">
    <tr>
      <td><xsl:value-of select="Name"/></td>
      <td><xsl:value-of select="Telefon"/></td>
    </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template></xsl:stylesheet>
```

Variante 2 (*Tabellenansicht.xsl*):

```
<?xml version="1.0" encoding="ISO-8859-1"?><xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="/">
  <html>
  <body>
    <h1>Telefonbuch</h1>
    <table border="1">
    <tr bgcolor="#EEEEEE">
      <th align="left">Name</th>
      <th align="left">Telefonnummer</th>
    </tr>
    <xsl:for-each select="Adressbuch/Adresse">
    <tr>
      <td><xsl:value-of select="Name"/></td>
      <td><xsl:value-of select="Telefon"/></td>
    </tr>
    </xsl:for-each>
    </table>
  </body>
  </html>
</xsl:template></xsl:stylesheet>
```

Verwendung und Test

Speichern Sie die obigen Dokumente im Webverzeichnis und fügen Sie zwei XML-Controls in eine Tabelle ein:



Weisen Sie in beiden Fällen der Eigenschaft *DocumentSource* den Wert "~/daten.xml" zu. *TransformSource* weisen Sie jeweils die Namen der Transformationsdateien zu.

Zur Laufzeit dürfte sich Ihnen der folgende Anblick bieten:

Wie Sie sehen, kann durch einfachen Austausch der Transformationsdateien ein gänzlich unterschiedliches Ergebnis erreicht werden (natürlich könnten Sie auch andere Filterbedingungen setzen etc.).

40.1.14 FileUpload

Nicht in jedem Fall muss man gleich mit Kanonen auf Spatzen schießen. Geht es um den Upload braucht nicht gleich ein FTP-Client eingesetzt zu werden, wenn Sie nur ein paar Dateien hochladen wollen.

Genau für diesen Einsatzzweck ist das *FileUpload*-Steuerelement gedacht. Hauptfunktion ist zunächst die Auswahl von Dateien per Dateidialog und Anzeige der Dateinamen (die Eigenschaft *AllowMultiple* steuert, ob Sie mehr als eine Datei auswählen können):

Die Funktion zum eigentlichen Upload müssen Sie über einen weiteren Button auslösen:

```
protected void Button8_Click(object sender, EventArgs e)
 {
    if (FileUpload1.HasFiles)  // früher nur HasFile
       {
           string rootPath = Server.MapPath("~/App_Data/");
           foreach (HttpPostedFile file in FileUpload1.PostedFiles)
           {
               file.SaveAs(Path.Combine(Server.MapPath("Dateien\\"), file.FileName));
               Label1.Text += String.Format("{0}<br />", file.FileName);
           }
       }
 }
```

HINWEIS: Der Zielpfad (in diesem Fall *Dateien*) muss für den ASP.NET-User Schreibrechte aufweisen, auch die maximale Dateigröße ist beschränkt.

40.1.15 AdRotator

Fluch oder Segen? Als Programmierer sollten Sie ruhig ein Auge auf dieses Control werfen, auch wenn Sie sonst Werbung aus den Webseiten herausfiltern.

Dreh- und Angelpunkt sind natürlich die Grafiken (ja, ja, die beliebten Banner) und eine Steuerdatei im XML-Format. Diese weisen Sie über die Eigenschaft *AdvertisementFile* zu.

BEISPIEL 40.19: Steuerdatei *Werbung.xml*

XML

```
<Advertisements>
   <Ad>
      <ImageUrl>banner1.gif</ImageUrl>
      <NavigateUrl>http://www.doko-buch.de</NavigateUrl>
      <AlternateText>Alt Text</AlternateText>
      <Keyword>Bücher</Keyword>
      <Impressions>80</Impressions>
   </Ad>
   <Ad>
      <ImageUrl>banner2.gif</ImageUrl>
      <NavigateUrl>http://www.doko-buch.de</NavigateUrl>
      <AlternateText>Alt Text</AlternateText>
      <Keyword>Bücher</Keyword>
      <Impressions>80</Impressions>
   </Ad>
   <Ad>
...
   </Ad>
</Advertisements>
```

Neben der Grafik geben Sie jeweils die Zieladresse, einen alternativen Text, Schlüsselwörter und die Wichtung des einzelnen Eintrags vor.

Das Ergebnis der Bemühungen:

Und nach einem Refresh:

HINWEIS: Alternativ können die Daten auch aus einer Datenquelle kommen.

40.2 Steuerelemente für die Seitennavigation

ASP.NET bietet für die Navigation zwischen Webseiten fast alles, was das Programmiererherz begehrt:

- ein *Menu*-Steuerelement, das im Handling fast sein Windows Forms-Pendant erreicht,
- ein *TreeView*-Steuerelement, das sowohl für die Navigation als auch für die Anzeige von XML-Daten oder anderen Datenquellen dient
- und ein *SiteMapPath*-Steuerelement, das dem Nutzer hilft, die Übersicht zu behalten.

HINWEIS: Für alle drei Steuerelemente gilt, dass nur im Zusammenhang mit einer Masterpage eine sinnvolle Navigation möglich ist. Andernfalls müssten Sie die Steuerelemente in jeder einzelnen Seite einfügen, was einen riesigen Aufwand bedeutet.

Last but not least, haben wir in diesen Abschnitt auch das *MultiView*-/*View*- und das *Wizard*-Steuerelement aufgenommen, obwohl diese nicht direkt der Navigation zwischen Seiten, sondern eher der Navigation zwischen Inhalten dienen. Der Übersicht über die Fülle der angebotenen Steuerelemente wird diese Klassifizierung sicherlich nicht abträglich sein.

Bevor wir jedoch zu den einzelnen Controls kommen, müssen wir etwas vom Thema abschweifen und uns um eine recht unscheinbare Datei kümmern, die jedoch für die Seitennavigation von zentraler Bedeutung ist.

40.2.1 Mehr Übersicht mit Web.Sitemap

Diese Datei, die im Web-Projektverzeichnis gespeichert wird, stellt die Zusammenhänge und Gliederungen zwischen den einzelnen Webseiten (ASPX) her. Wie nicht anders zu erwarten,

handelt es sich um eine XML-Datei mit dem Root-Element *siteMap*. Diesem ist genau **ein** *siteMapNode*-Element unterzuordnen, es handelt sich um die Startseite Ihrer Webanwendung. Unter diesem *siteMapNode*-Element können Sie hierarchisch weitere Elemente einordnen.

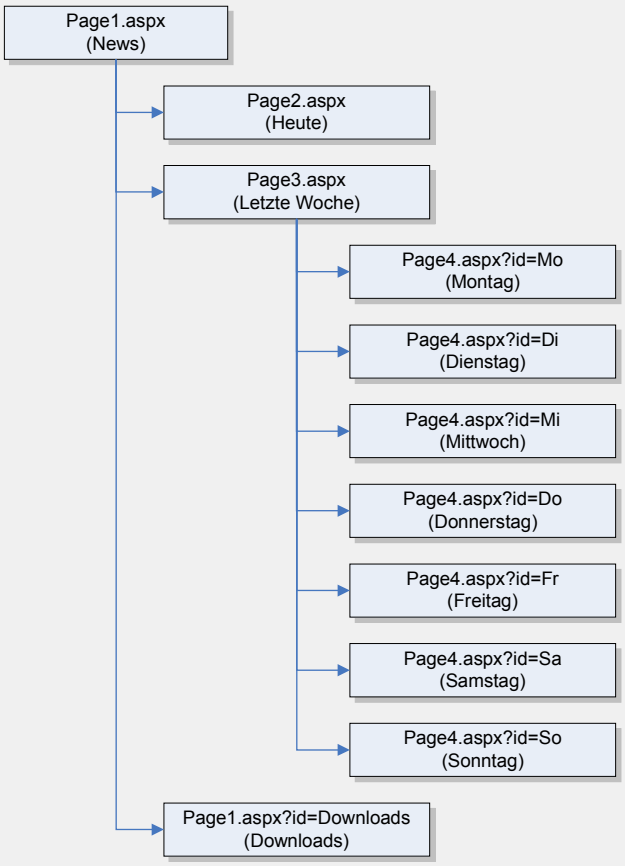
Die Attribute der einzelnen Element beschreiben

- die verknüpfte Seite (*url*),
- den anzuzeigenden Titel (*title*)
- sowie den Tooltip (*description*).

Alternativ können auch Ressourcen- und Security-Informationen in den einzelnen Elementen angelegt werden.

BEISPIEL 40.20: Um folgende Seitenstruktur zu beschreiben ...

Ergebnis



XML

... verwenden wir diese *Web.Sitemap*

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
```

BEISPIEL 40.20: Um folgende Seitenstruktur zu beschreiben ...

```
    <siteMapNode url="page1.aspx" title="News"  description="News für die Leser">
        <siteMapNode url="page2.aspx" title="Heute"  description="" />
        <siteMapNode url="page3.aspx" title="Letzte Woche"  description="" >
        <siteMapNode url="page4.aspx?id=Mo" title="Montag"  description="" />
        <siteMapNode url="page4.aspx?id=Di" title="Dienstag"  description="" />
        <siteMapNode url="page4.aspx?id=Mi" title="Mittwoch"  description="" />
        <siteMapNode url="page4.aspx?id=Do" title="Donnerstag"  description="" />
        <siteMapNode url="page4.aspx?id=Fr" title="Freitag"  description="" />
        <siteMapNode url="page4.aspx?id=Sa" title="Samstag"  description="" />
        <siteMapNode url="page4.aspx?id=So" title="Sonntag"  description="" />
      </siteMapNode>
  <siteMapNode url="page1.aspx?id=Downloads" title="Downloads"  description="" />
 </siteMapNode>
</siteMap>
```

Wie erstelle ich so eine Datei? Diese entscheidende Frage müssen wir mit der wohl nicht ganz befriedigenden Aussage "mit einem Texteditor" beantworten. Ein eigener Assistent ist auch in der aktuellen Visual Studio Version nicht enthalten.

Sollen Steuerelemente an die obige Datei gebunden werden, nutzen Sie einfach ein *SiteMapDataSource*-Control. Dieses muss nicht weiter konfiguriert werden, da zunächst davon ausgegangen wird, dass in jedem Projekt nur eine Sitemap enthalten ist. Auf optionale Parameter und deren Auswirkung gehen wir im Zusammenhang mit den im Folgenden vorgestellten Steuerelementen ein.

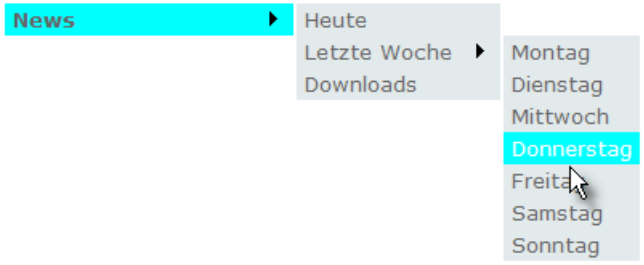
40.2.2 Menu

Das *Menu*-Steuerelement stellt die klassische Menüleiste auch für Ihre Webprojekte zur Verfügung. Das Menü gliedert sich in zwei wesentliche Teile:

- Der statische, dauernd angezeigte Teil des Menüs
- und der der dynamische, durch Maus- oder Tastatur aktivierte Teil.

Menüs können Sie sowohl horizontal (*Orientation=Horizontal*):

als auch vertikal (*Orientation=Vertical*) ausrichten:

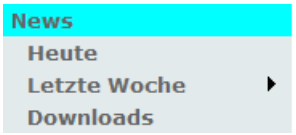


HINWEIS: Beide Abbildungen zeigen die im Abschnitt 40.2.1 definierte SiteMap.

Datenbindung an eine SiteMap

Wie schon erwähnt, genügt für die Datenbindung ein *SiteMapDataSource*-Steuerelement, das Sie über die Eigenschaft *DataSourceId* an das Menu-Steuerelement binden. Nachfolgend wird bereits das erste *siteMapNode*-Element aus der Datei *Web.Sitemap* angezeigt.

Doch ach, an dieser Stelle scheiden sich die Geister an dem, was denn nun als statisches Menü angezeigt werden soll. Die Beschränkung auf einen übergeordneten *siteMapNode* können wir nicht aufheben, auf den Sinn kommen wir noch später zurück. Doch über die Eigenschaft *StaticDisplayLevels* können wir weitere Menüebenen in den statischen Anzeigebereich aufnehmen:



Alternativ können Sie auch die *SiteMapDataSource* manipulieren, setzen Sie hier die Eigenschaft *ShowStartingNode* auf *False*, wird die nachfolgende Menüebene als statisches Menü angezeigt:

HINWEIS: Doch Vorsicht! So nehmen Sie dem Anwender die Möglichkeit, über das Menü zur Startseite zurückzukehren.

Ist die Bindung an die *SiteMapDataSource* erfolgt, und haben Sie die erforderlichen optischen Anpassungen vorgenommen, ist Ihr Menü bereits voll einsatzbereit. Tooltipps und Zieladressen der

einzelnen Menüeinträge werden aus der Datei *Web.Sitemap* entnommen, Ereignisse brauchen Sie nicht zu verarbeiten.

Menü ohne Datenbindung

Alternativ können Sie ein Menü auch ohne Datenbindung zur Programmsteuerung verwenden. Neue Menüeinträge erzeugen Sie zur Laufzeit über die *Add*-Methode des *Items*- bzw. des *ChildItems*-Objekts.

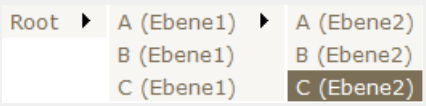
BEISPIEL 40.21: Erzeugen einer Menüstruktur

C#

```
protected void Button1_Click(object sender, EventArgs e)
{
  Menu3.Items.Add(new MenuItem("Root", "", "", "~//page1.aspx"));

  Menu3.Items[0].ChildItems.Add(new MenuItem("A (Ebene1) ", "", "", "~//page2.aspx"));

  Menu3.Items[0].ChildItems[0].ChildItems.Add(new MenuItem("A (Ebene2) ", "", "",
              "~//page2.aspx"));
  Menu3.Items[0].ChildItems[0].ChildItems.Add(new MenuItem("B (Ebene2) ", "", "",
              "~//page2.aspx"));

  Menu3.Items[0].ChildItems[0].ChildItems.Add(new MenuItem("C (Ebene2) ", "", "",
              "~//page2.aspx"));

  Menu3.Items[0].ChildItems.Add(new MenuItem("B (Ebene1)", "", "", "~//page2.aspx"));
  Menu3.Items[0].ChildItems.Add(new MenuItem("C (Ebene1)", "", "", "~//page2.aspx"));
}
```

Ergebnis

Wem hier die Übersicht verloren gegangen ist, der findet sich mit Hilfe der folgenden Abbildung zurecht, die das Endergebnis zeigt:



HINWEIS: Von den angebotenen Überladungen der *Add* -Methode haben wir uns für die Möglichkeit entschieden, direkt einen Navigations-Url anzugeben.

Haben Sie keinen Navigations-Url angegeben, können Sie im *MenuItemClick*-Ereignis die Nutzerauswahl über den Parameter *e.Item* auswerten.

BEISPIEL 40.22: Anzeige des *Value*-Wertes

C#

```
protected void Menu3_MenuItemClick(object sender, MenuEventArgs e)
{
  Label1.Text = e.Item.Value;
```

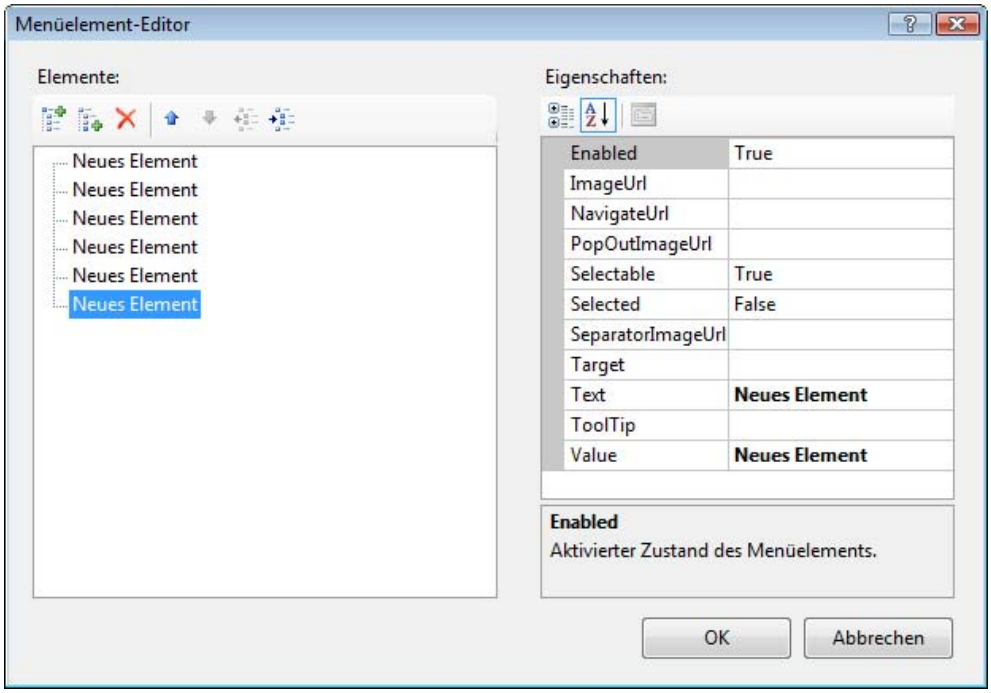
BEISPIEL 40.22: Anzeige des *Value*-Wertes

```
}
```

Alternativ können Sie sich auch den kompletten Pfad bis zum aktuellen Menüeintrag ausgeben lassen:

```
Label1.Text = e.Item.ValuePath;
```

HINWEIS: Natürlich können Sie das Menü auch direkt per Assistent in Visual Studio 2012 erstellen. Nutzen Sie dazu entweder das Kontextmenü oder die Eigenschaft *Items*.



HINWEIS: Auf die diversen Möglichkeiten, das Menü optisch anzupassen, gehen wir hier nicht ein, dafür finden Sie in der Hilfe genügend Unterstützung.

40.2.3 TreeView

Mit ASP.NET steht Ihnen auch ein *TreeView*-Steuerelement zur allgemeinen Verfügung. Über Sinn und Zweck dieses Steuerelements brauchen wir sicher nicht lange zu philosophieren, wesentlichste Aufgabe ist die hierarchische Darstellung von Informationen. Ob Sie das Control zur Navigation zwischen den einzelnen Seiten oder für die reine Programmsteuerung nutzen, bleibt Ihnen überlassen.

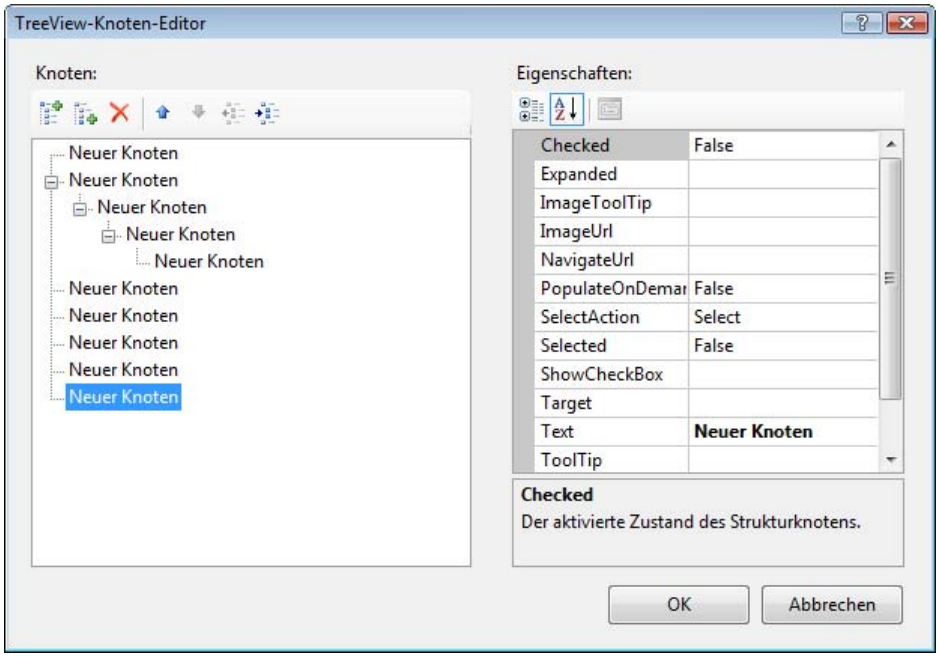
Wie auch beim *Menu*-Steuerelement können Sie das Control mit oder ohne Datenbindung betreiben. Geht es beispielsweise um die statische Darstellung der SiteMap, binden Sie das Control einfach an ein *SiteMapDataSource*-Control.

BEISPIEL 40.23: *TreeView* an *SiteMapDataSource*-Control gebunden

Auch hier steht Ihnen automatisch der URL für die Navigation zur Verfügung, eine eine Programmierung ist nicht erforderlich.

TreeView ohne Datenbindung

TreeView-Einträge erzeugen Sie entweder per Code oder mit dem *TreeView-Knoten-Editor*, den Sie über das Kontextmenü oder die Eigenschaft *Nodes* erreichen:



Die Eigenschaft *Nodes* verwenden Sie auch, wenn Sie die Einträge per Code erzeugen. Hier stehen Ihnen die Methoden *Add* und *AddAt* zur Verfügung, deren Verwendung Ihnen ein kleines Beispiel zeigen soll.

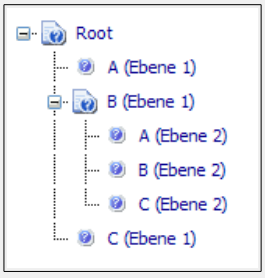
BEISPIEL 40.24: *TreeNodes* erzeugen

C#

```
protected void Button2_Click(object sender, EventArgs e)
{
  TreeView2.Nodes.Add(new TreeNode("Root"));
  TreeView2.Nodes[0].ChildNodes.Add(new TreeNode("A (Ebene 1)", "Value A"));
  TreeView2.Nodes[0].ChildNodes.Add(new TreeNode("B (Ebene 1)", "Value B"));
  TreeView2.Nodes[0].ChildNodes.Add(new TreeNode("C (Ebene 1)", "Value C"));
  TreeView2.Nodes[0].ChildNodes[1].ChildNodes.Add(new TreeNode("A (Ebene 2)", "Value BA"));
  TreeView2.Nodes[0].ChildNodes[1].ChildNodes.Add(new TreeNode("B (Ebene 2)", "Value BB"));
  TreeView2.Nodes[0].ChildNodes[1].ChildNodes.Add(new TreeNode("C (Ebene 2)", "Value BC"));
}
```

Ergebnis

Der erzeugte Baum:



BEISPIEL 40.25: *TreeNodes* zur Laufzeit erzeugen

C#

Kommen die Daten aus einer Datenbank, verwenden Sie sinnvollerweise einen *DataReader*.

```
if (!IsPostBack)
{
  OleDbConnection conn = new OleDbConnection(Connstr);
  OleDbCommand cmd = new OleDbCommand("SELECT * FROM artikelgruppen", conn);
  conn.Open();
  OleDbDataReader dr = cmd.ExecuteReader();
  while (dr.Read())
  {
    TreeNode tn = new TreeNode();
    tn.Text = dr["Name"].ToString();
    tn.Value = dr["id"].ToString();
    TreeView1.Nodes[0].ChildNodes.Add(tn);
  }
  dr.Close();
  conn.Close();
```

Doch nicht in jedem Fall möchte man umfangreiche hierarchische Strukturen gleich komplett in die *TreeView* einlesen (z.B. Verzeichnisse). Dafür bietet sich ein ereignisorientierter Ansatz mit dem Event *TreeNodePopulate* an.

BEISPIEL 40.26: Verwendung von *TreeNodePopulate*

C#

Erzeugen Sie beispielsweise mit dem Knoten-Editor einen neuen Eintrag und setzen Sie dessen *PopulateOnDemand*-Eigenschaft auf *True*. Weiterhin muss die Eigenschaft *TreeView.ExpandDepth* auf 0 gesetzt werden, da anderenfalls der komplette Baum eingelesen wird.

Der Ereigniscode wird durch das Expandieren des entsprechenden *TreeNode* ausgelöst:

```
protected void TreeView2_TreeNodePopulate(object sender, TreeNodeEventArgs e)
{
  e.Node.ChildNodes.Add(new TreeNode("A"));
  e.Node.ChildNodes.Add(new TreeNode("B"));
  e.Node.ChildNodes.Add(new TreeNode("C"));
  e.Node.ChildNodes.Add(new TreeNode("D"));
}
```

Die Auswahl des Nutzers können Sie über das *SelectedNodeChanged*-Ereignis auswerten:

```
protected void TreeView2_SelectedNodeChanged(object sender, EventArgs e)
{
  Label1.Text = TreeView2.SelectedNode.Text;
}
```

Auch dieses Control bietet eine schon fast unüberschaubare Fülle an Eigenschaften, die wir an dieser Stelle beim besten Willen nicht alle vorstellen können.

40.2.4 SiteMapPath

Allen von Windows Forms kommenden Umsteigern wird dieses Steuerelement zunächst völlig unbekannt vorkommen. Der Zweck dieses Controls besteht darin, dem Nutzer den aktuellen Stand in einer hierarchischen Seitenstruktur zu vermitteln und eine schnelle Navigationshilfe zu übergeordneten Seiten anzubieten. Dabei handelt es sich immer nur um den Weg zurück, wie z.B. im Märchen "Hänsel und Gretel" (dort wurden von den Kindern Brotkrumen gestreut, um den Weg zurück zu finden). Der Begriff "Brotkrumen" (*Breadcrumb*) taucht auch im Zusammenhang mit diesem Control auf.

Doch nun zur Realisierung und Einbindung.

Voraussetzung für die Verwendung des Steuerelements ist lediglich eine erstellte SiteMap (*Web.SiteMap*), Datenbindungen oder andere Konfigurationen sind an dieser Stelle nicht erforderlich.

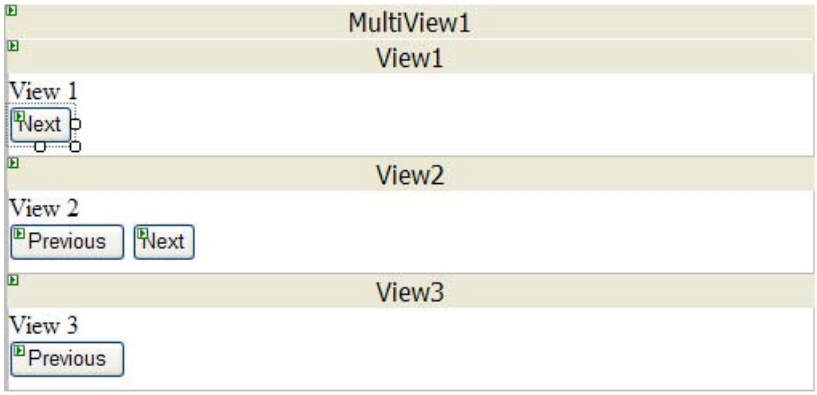
Nach dem Start (sinnvollerweise binden Sie ein *Menu* oder eine *TreeView* ein, um auch zwischen den Seiten zu navigieren) können Sie bereits die Funktionsweise überprüfen:

Ein Klick auf *News* bzw. *Letzte Woche* bringt Sie sofort wieder auf die gewählte Seite zurück, das *Menu* bzw. die *TreeView* zeigen den entsprechenden Eintrag an.

HINWEIS: Auch die direkte Eingabe einer Adresse im Browser bringt das Control nicht aus dem Tritt, es wird der korrekte Pfad zu dieser Datei angezeigt.

40.2.5 MultiView, View

Mit dem neuen *MultiView*-Steuerelement steht auch dem ASP.NET-Programmierer die Funktionalität eines *TabControls* (WinForms) – jedoch ohne dessen Tabulatoren – zur Verfügung.



Innerhalb des *MultiView* können beliebig viele *View*-Steuerelemente platziert werden, von denen nur eines aktiv sein kann und zum Client übertragen wird. Die Navigation zwischen den Seiten lässt sich mit verschiedenen Varianten bewerkstelligen.

- Sie verwenden die Eigenschaft *MultiView.ActiveViewIndex* und schalten beispielsweise per *ComboBox* oder über einen *Button* zur gewünschte *View*.
- Sie platzieren in den Views zusätzliche Buttons und nutzen deren Eigenschaft *CommandName* bzw. *CommandArgument*, um Steuerkommandos an die *MultiView* abzusetzen. Dabei können Sie zwischen absoluter und relativer Navigation wählen.

BEISPIEL 40.27: ***MultiView.ActiveViewIndex***

C#
```
protected void Button2_Click(object sender, EventArgs e)
{
  MultiView1.ActiveViewIndex = 1;
}
```

oder

```
protected void DropDownList1_SelectedIndexChanged(object sender, EventArgs e)
{
  MultiView1.ActiveViewIndex = DropDownList1.SelectedIndex;
}
```

BEISPIEL 40.28: Absolute Navigation

HTML
```
<asp:Button ID="Button4" runat="server"  Text="View2"
            CommandArgument="1" CommandName="SwitchViewByIndex"
/>
```

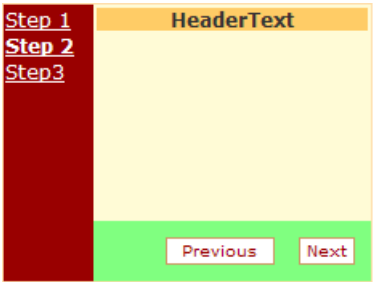
BEISPIEL 40.29: Relative Navigation

HTML
```
<asp:Button ID="Button5" runat="server" Text="Previous" CommandName="PrevView" />
<asp:Button ID="Button7" runat="server" Text="Next" CommandName="NextView" />
```

40.2.6 Wizard

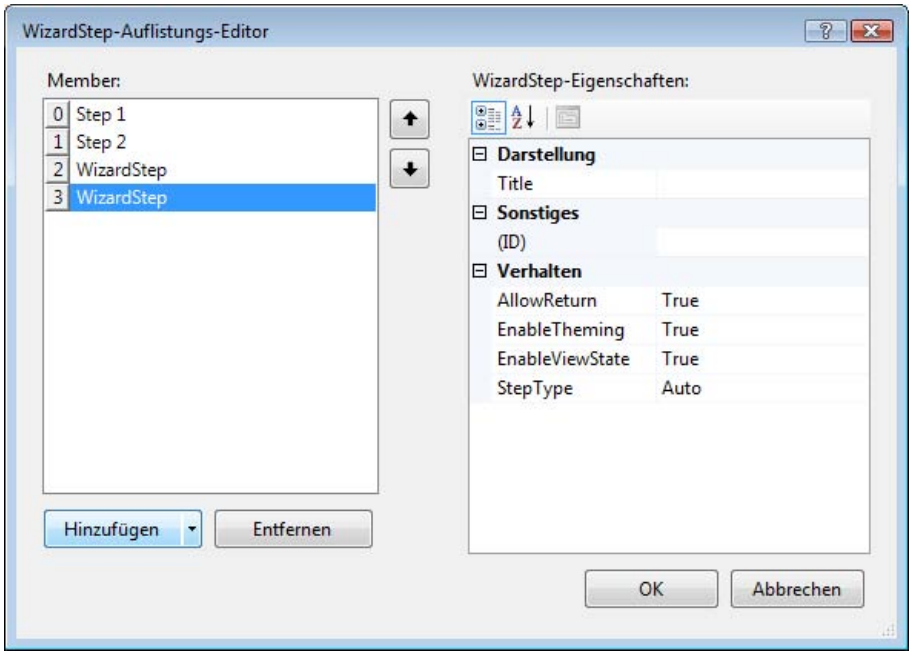
Auch das *Wizard*-Steuerelement dient der Navigation zwischen Inhalten, speziell für Vorgänge, die in mehreren und teilweise voneinander abhängigen Schritten ausgeführt werden müssen. Man denke dabei nur an einen Registrierungs-Assistenten oder an eine Bestellung (Adresse, Lieferadresse, AGB's etc.) in einem Webshop.

Die Funktionsweise lässt sich am einfachsten mit der eines WinForm *TabControl* vergleichen, auch wenn der Funktionsumfang wesentlich größer ist.



Die Funktionalität zum Wechsel zwischen den einzelnen Wizard-Schritten steht bereits zur Entwurfszeit zur Verfügung, einem echten WYSIWYG-Entwurf steht also nichts im Weg.

Erzeugen Sie also zunächst mit dem WizardStep-Auflistungs-Editor (Eigenschaft *WizardSteps*) so viele Schritte wie Sie benötigen:



Nachfolgend können Sie in die einzelnen Ansichten die gewünschten Steuerelemente einfügen.

Die Ablauflogik für die Navigation zwischen den einzelnen Ansichten ist bereits implementiert, eine wichtige Eingriffsmöglichkeit bietet in diesem Zusammenhang die Eigenschaft *StepType*, mit der Sie zwischen

- *Auto* (Buttons werden automatisch bereitgestellt),
- *Start* (erste Seite),
- *Step* (normaler Zwischenschritt) ,
- *Finish* (letzter Schritt mit Finish-Button),
- *Complete* (letzte Anzeige, ohne Buttons)

unterscheiden können. Über *AllowReturn* verhindern Sie, dass ein Nutzer vorhergehende Schritte aufrufen kann.

Interessant ist auch das Ereignis *ActiveStepChanged*, mit dem Sie beispielsweise verhindern können, das Schritte einfach übergangen werden.

BEISPIEL 40.30: Eine *CheckBox* auf Seite 3 (!) muss angeklickt werden, bevor es zur Seite 4 geht.

C#
```
protected void Wizard1_ActiveStepChanged(object sender, EventArgs e)
{
  if ((Wizard1.ActiveStepIndex == 3) & (!CheckBox1.Checked))
    Wizard1.ActiveStepIndex = 2;
}
```

Alternativ können Sie auch mit *GetHistory* auswerten, welche Schritte der Nutzer bearbeitet hat.

BEISPIEL 40.31: Auflisten aller Schritte

C#
```
foreach (WizardStep step in Wizard1.GetHistory())
  Response.Write(step.Name+ "<br>");
```

Ergebnis

Die Ausgabe beim "normalen" Ablauf:

```
Fertig
Step 3
Step 2
Step 1
```

40.3 Webseitenlayout/-design

Der folgende Abschnitt wird sich mit "Oberflächlichkeiten", d h. mit Verfahren zur effizienten und einheitlichen Gestaltung der Weboberfläche, auseinandersetzen. Zwei wesentliche Technologien bieten sich dafür in ASP.NET an:

- Masterpages,
- Themes.

Beide bieten dem Programmierer geeignete Lösungen, um mit relativ wenig Aufwand, Seiten einheitlich zu gestalten.

40.3.1 Masterpages

Geht man von einer "normalen" Webanwendung aus, sollten die einzelnen Seiten möglichst das gleiche Layout (Navigation, Werbung, Impressum etc.) bei unterschiedlichen Inhalten aufweisen. Eine Lösungsmöglichkeit wäre die Verwendung von Frames. Dagegen spricht jedoch, dass zum einen die Unterstützung durch die Clients (Browser) nicht einheitlich ist, zum anderen ist die Steuerung in ASP.NET nur suboptimal.

Besser und wesentlich einfacher programmierbar sind Tabellen zur Gliederung der Seite (Navigationsbereich, Kopfzeile, Content ...). Doch hier lauert auf den Programmierer viel Arbeit, müssen doch auf jeder Seite gleiche Inhalte mehrfach programmiert werden. Änderungen können dann schnell im Chaos enden.

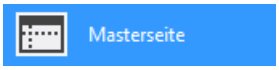
An dieser Stelle setzt das Konzept der Masterpages an. Die Masterpage definiert die übergreifenden Elemente einer Website und bietet quasi einen Platzhalter (*ContentPlaceHolder*) für den Inhalt den die einzelnen Seiten darstellen sollen.

Masterpage

Erstellen Sie zunächst wie gewohnt ein Webprojekt und löschen Sie die *Default.aspx*.

HINWEIS: Bevor Sie Content-Seiten erstellen können, muss die Masterpage erzeugt werden.

Über den Menüpunkt *Website|Neues Element hinzufügen* fügen Sie jetzt eine Masterpage hinzu:

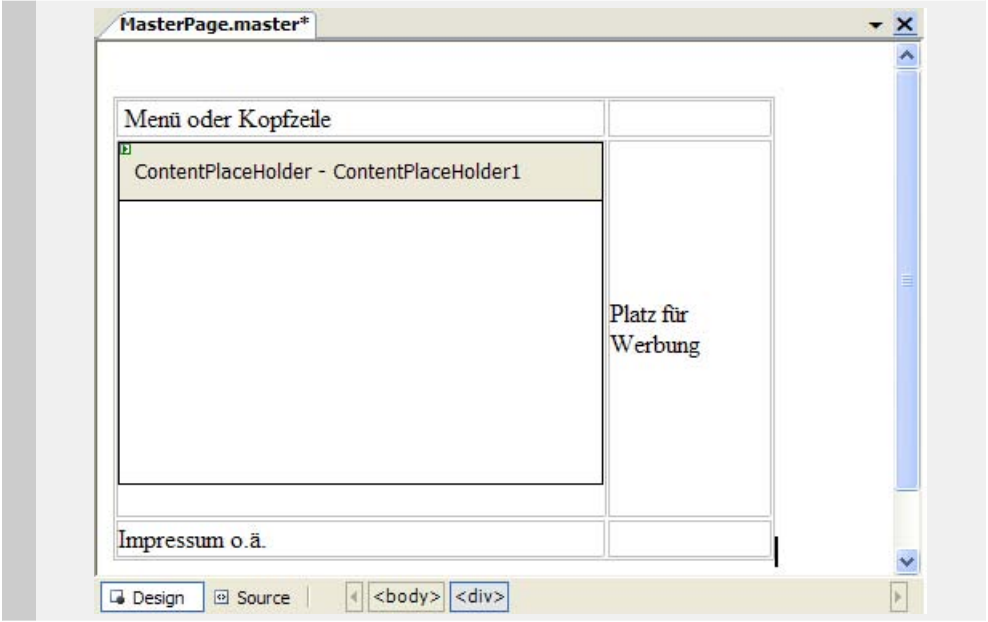


Sie erhalten eine normale WebForm-Seite, in der sich ein *ContentPlaceHolder*-Steuerelement befindet. Dieses verfügt über fast keine Eigenschaften, es soll lediglich den Bereich markieren, in welchen später die Inhalte der Contentseiten eingeblendet werden.

Im Unterschied zu den einfachen Web Formularen ist die Extension für diese Seite mit *.master* festgelegt.

HINWEIS: Es ist günstig, den *ContentPlaceHolder* in eine HTML-Table einzufügen, so können Sie Breite und Position einfach bestimmen.

BEISPIEL 40.32: Masterpage mit *ContentPlaceHolder*

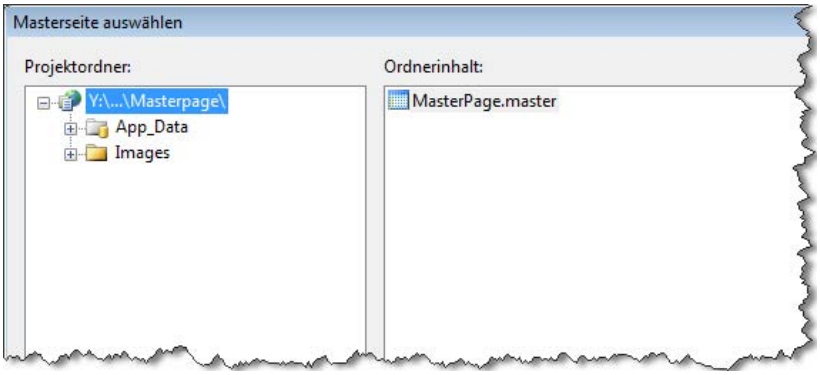


HINWEIS: Auf einer Masterpage können Sie mehrere *ContentPlaceHolder* einfügen, was zum Beispiel bei spaltenorientierten Webseiten recht nützlich ist.

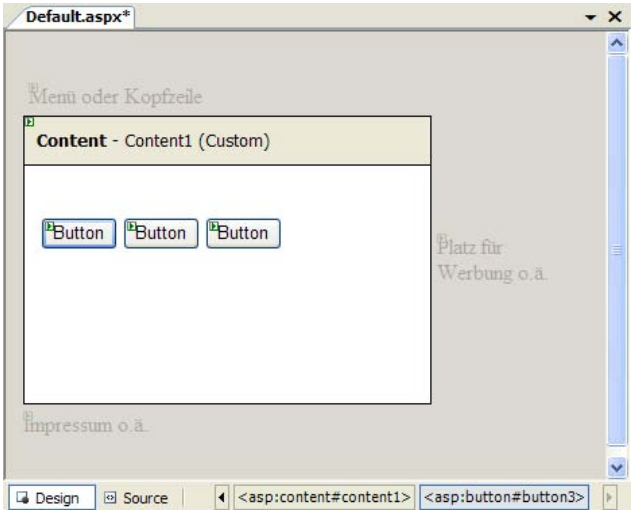
Last but not least können Sie Masterpages auch ineinander verschachteln, was der Übersicht des Programmierers allerdings wenig zuträglich ist.

Content Seiten

Erstellen Sie diese wie normale Web Forms, beachten Sie jedoch, dass Sie im Assistenten die Option "Master Page auswählen" markieren. In einem zweiten Schritt erhalten Sie die Möglichkeit, der Seite eine Masterpage zuzuordnen.



Der Seitenbereich beschränkt sich jetzt auf den Bereich, der in der Masterpage vom *ContentPlaceHolder* reserviert wurde (die Höhe passt sich jedoch dem Inhalt an). Alle in der Masterpage enthaltenen Steuerelemente werden visuell dargestellt, können jedoch nicht verändert werden (was sicher wenig sinnvoll wäre).



Fügen Sie in den *Content*-Bereich beliebige Steuerelemente ein, an der prinzipiellen Programmierung hat sich nichts geändert. Mit einer Ausnahme: Wollen Sie aus dem Code heraus auf die Steuerelemente der Masterpage zugreifen, müssen Sie sich auch auf die Masterpage beziehen (Objekt *Master*). So einfach wie bei den WinForms ist es allerdings nicht, Sie können enthaltene Steuerelemente nicht direkt, sondern nur über *FindControl* ansprechen.

BEISPIEL 40.33: Ändern einer *Label*-Beschriftung in der Masterpage

C#
```
protected void Button1_Click(object sender, EventArgs e)
{
  (Master.FindControl("label1") as Label).Text = "Hier spricht die Contentpage";
}
```

HINWEIS: Mehr zu Content Seiten finden Sie im Praxisbeispiel 40.5.2.

40.3.2 Themes/Skins

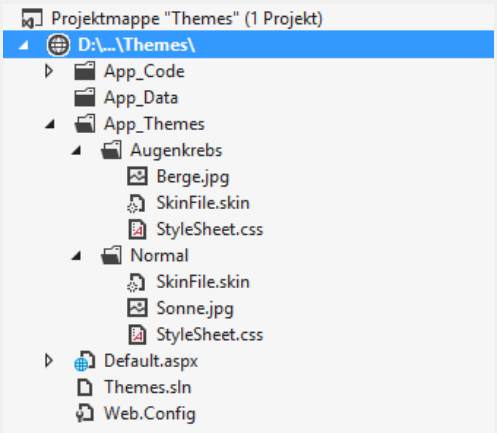
Möchten Sie Ihre Webseiten einheitlich gestalten und eventuell dem Nutzer die Wahl verschiedener Layouts überlassen? Wenn ja, dann sind Sie hier richtig.

Mittels so genannter Themes können Sie seitenübergreifende Vorgaben für die optische Gestaltung Ihrer Website realisieren. Ein Theme besteht dabei aus folgenden optionalen Elementen:

- Skin-Dateien (*.skin*),
- Cascading Style Sheets (*.css*)
- und natürlich diversen Grafiken (*.jpg*, *.gif* ...),

die in einem Unterverzeichnis (Name=Themename) gesammelt werden. Diese Theme-Verzeichnisse werden wiederum im Unterordner *App_Themes* des Projekts gesammelt.

BEISPIEL 40.34: Projekt mit zwei Themes



Zuweisen des aktiven Theme

Drei Varianten der Zuweisung eines Theme bieten sich an:

- global über die *Web.Config* des aktuellen Projekts (*<pages theme="<themename>">*),
- über die Seitendirektive (*@Page ... theme="<themename>"*)
- und nicht zuletzt über den Programmcode (*Page.Theme="<themename>"*).

Ob auf ein Steuerelement auch ein Theme angewendet wird, bestimmen Sie mit der Eigenschaft *EnableTheming*.

BEISPIEL 40.35: *Web.Config*

XML

```
<configuration
    <appSettings/>
    <connectionStrings/>
    <system.web>
        <pages theme="Theme1"></pages>
```

BEISPIEL 40.36: Seitendirektive

HTML

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeFile="Default.aspx.cs" Inherits="_Default" Theme="Theme1"%>
```

BEISPIEL 40.37: Programmcode

C#

```
...
public partial class _Default : System.Web.UI.Page
{
 void Page_PreInit(object sender, EventArgs e)
 {
      Page.Theme = "Theme1";
...
```

Skin-Datei

Diese Datei enthält die Formatierungsbeschreibungen für die einzelnen Steuerelemente. Wenn von "einzelne" die Rede ist, sollten Sie das wörtlich nehmen, für jedes Steuerelement (das Sie in Ihren Webseiten verwenden) ist ein Eintrag vorzunehmen. Anderenfalls werden die Standardeinstellungen genommen.

HINWEIS: Die Skin-Datei fügen Sie über das Kontextmenü des Projektmappen-Explorers dem jeweiligen Theme hinzu.

BEISPIEL 40.38: Skin-Einstellung für alle TextBoxen

HTML

```
<asp:TextBox runat="server"
             Style="font-weight: bold; color: #ffff00;
             border-top-style: none; border-right-style: none;
             border-left-style: none; background-color: #66ff33;
             border-bottom-style: none"></asp:TextBox>
```

HINWEIS: Wollen Sie für **ein** Steuerelement mehrere Layouts in einem Skin ablegen, fügen Sie ein Attribut *skinid* hinzu. Über dieses entscheidet das spätere Steuerelement, welches Layout gewünscht ist (siehe Praxisbeispiel 40.5.1).

StyleSheet

Über diese Datei, die auch dem ASP-Programmierer bekannt vorkommen wird, können Sie das Aussehen auf HTML-Ebene beeinflussen.

BEISPIEL 40.39: Ändern des Aussehens der Überschriftenebenen H1 und H2

HTML

```
h1 {
    font-size: 20pt;
    margin-bottom: 18pt;
    color: #0033ff;
    font-family: Arial;}

h2 {
    font-size: 18pt;
    margin-bottom: 18pt;
    color: #00ffcc;
    font-family: Arial;}
```

Grafiken

Auch für Grafiken trifft dasselbe wie für Skin-Dateien zu. Möchten Sie beispielsweise ein *Image*-Steuerelement in Abhängigkeit des Theme mit unterschiedlichen Grafiken beschicken, genügt es, wenn Sie die Skin-Datei um einen Eintrag für das *Image*-Steuerelement erweitern.

HINWEIS: Vergessen Sie nicht eine *SkinId* zu vergeben, sonst werden die Grafiken in allen *Image*-Steuerelementen ersetzt.

BEISPIEL 40.40: Grafik im Skin-File zuweisen

HTML

```
<asp:Image runat="server"
           ImageUrl="~/App_Themes/Augenkrebs/Berge.jpg" SkinID="LOGO" />
```

BEISPIEL 40.40: Grafik im Skin-File zuweisen

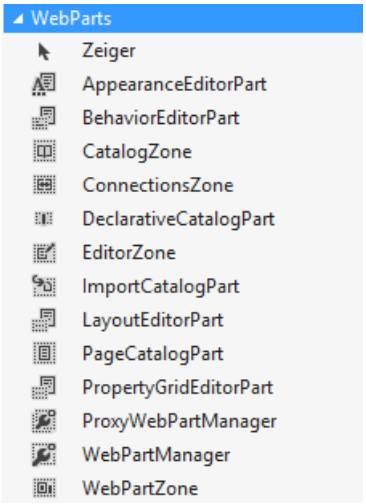
Die Imagedefinition im HTML-Quelltext:

```
<asp:Image ID="Image1" runat="server" SkinID="LOGO" />
```

Ähnlich können Sie auch mit Hintergrundgrafiken (Anpassung von *body* im Style Sheet) etc. verfahren.

40.3.3 Webparts

An dieser Stelle wollen wir uns schon vorsorglich entschuldigen, eine umfassende Darstellung des Themas würde den Rahmen dieses Kapitels sprengen und sollte der Spezialliteratur vorbehalten bleiben. Wir wollen trotzdem nicht darauf verzichten, Ihnen einen ersten Einblick zu gewähren, auch wenn allein die Liste der verfügbaren Steuerelemente davon kündet, um welch komplexes Thema es sich handelt:



Über Webparts können Sie die Anwender Ihrer Website in die Lage versetzen, einzelne Teile der Seite ihren Bedürfnissen anzupassen. Dies betrifft sowohl die Position/Sichtbarkeit als auch die Eigenschaften einzelner Webseitenbereiche (die eigentlichen Webparts). Webparts lassen sich jedoch nicht gänzlich frei positionieren, sondern nur in den dafür vorgesehenen Bereichen, den so genannten *WebPartZones*.

Wem das Ganze bekannt vorkommt, der liegt nicht falsch, wenn er an Portalseiten oder diverse Content Management Programme denkt, die nach diesem Konzept aufgebaut sind.

Selbstverständlich lassen sich Webparts auch mit den anderen Technologien (Masterpages, Themes), die in diesem Abschnitt bereits vorgestellt wurden, kombinieren, wir verzichten jedoch besser darauf, um den Überblick nicht ganz zu verlieren.

HINWEIS: Das Praxisbeispiel 40.5.3 demonstriert Ihnen die Vorgehensweise und die wichtigsten Techniken im Zusammenhang mit Webparts.

40.4 Die Validator-Controls

Bei unseren bisherigen Experimenten sind wir vom Idealfall des Webanwenders ausgegangen: ein Nutzer, der immer alle Felder korrekt ausfüllt, keine falschen Werte eingibt und auch logisch alles richtig macht. Leider werden wir im täglichen Programmiererdasein fast immer nur auf DAUs[1] treffen, die mit traumwandlerischer Sicherheit genau die Fehler machen, auf die unsere Anwendung nicht vorbereitet ist.

In weiser Voraussicht haben die MS-Programmierer ein recht wirkungsvolles Werkzeug für die Eingabeprüfung entwickelt: die Validator-Controls.

40.4.1 Übersicht

Mit Hilfe der Validator-Controls bietet sich Ihnen eine einfache und vor allem flexible Möglichkeit, Benutzereingaben zu überprüfen. Neben einfachen Regeln, wie das Vorhandensein eines Eintrags, können Sie auch komplexe Vergleiche oder nutzerdefinierte Regeln realisieren.

Ein Validator-Control wird immer einem Eingabe-Control zugeordnet. Im Gegensatz dazu lassen sich einem Eingabe-Control auch mehrere Validator-Controls zuweisen. Zum Beispiel können Sie zunächst auf das Vorhandensein einer Kreditkartennummer prüfen, nachfolgend können Sie diese logisch prüfen.

Auf der "Validierung"-Seite des Werkzeugkastens finden Sie folgende Validator-Controls:

Control	Beschreibung
RequiredFieldValidator	Verknüpfen Sie dieses Control mit einer Textbox, muss diese einen Inhalt haben, bevor ein Post erfolgt.
CompareValidator	Dieses Control ermöglicht den Vergleich (=, >, <, >=, <=, <>) eines Eingabewertes mit einem Vorgabewert, der als Konstante oder Vergleichsfeld vorliegt.
RangeValidator	Der Inhalt des Controls muss mit einem vorgegebenen Bereich übereinstimmen.
RegularExpressionValidator	Der Inhalt des zu überwachenden Controls wird auf der Basis eines regulären Ausdrucks geprüft.
CustomValidator	Dieses Control können Sie verwenden, wenn Sie mit den bisher vorgestellten Controls nicht die gewünschte Prüfung vornehmen können. Sie können eine eigene Ereignisbehandlung realisieren.
ValidationSummary	Dieses Control fasst lediglich die Meldungen der o.g. Controls zusammen.

[1] Dümmster Anzunehmender User

40.4.2 Wo findet die Fehlerprüfung statt?

Grundsätzlich erfolgt die Prüfung der eingegebenen Daten auf dem Server und zwar mit den Regeln, die durch die Validator-Controls aufgestellt worden sind.

Je nach Typ des Browsers[1] kann auch eine erste Prüfung auf dem Client erfolgen, um unnötige Datenübertragungen zwischen Client und Server zu vermeiden. Dieses Feature wird mit der Eigenschaft *EnableClientScript* aktiviert, die Überprüfung auf Browsertyp und Browsereignung nehmen die Controls automatisch vor. Da bei diesem Verfahren zunächst keine Abfrage zum Server gesendet wird, ist ein besseres Antwortverhalten zu erwarten.

HINWEIS: Beachten Sie jedoch, dass eine endgültige Prüfung in jedem Fall auf dem Server erfolgt, egal ob die Prüfung auf dem Client bereits erfolgreich war[2].

Die Möglichkeit, Prüfungen unabhängig vom Browsertyp und dessen Fähigkeiten durchzuführen, vereinfacht die Entwicklung robuster Eingabemasken wesentlich.

40.4.3 Verwendung

Alle Validator-Controls sind zur Laufzeit zunächst unsichtbar, tappt der Nutzer jedoch in die von uns aufgestellte Falle und versucht, einen fehlerhaften Eingabewert an den Server zu senden, erscheint die gewünschte Fehlermeldung **im** Validator-Control (so wie Sie diese zur Entwurfszeit festgelegt haben).

Ein erster Test zeigt die Vorgehensweise: Platzieren Sie ein *RequiredFieldValidator*-Control auf dem Form und verbinden Sie dieses mit einer Textbox. Dazu genügt es, wenn Sie die Eigenschaft *ControlToValidate* auf das gewünschte Control festlegen. Wo Sie das Control platzieren, ist vollkommen egal, Sie sollten den Fehler jedoch nicht zwanzig Zeilen weiter unten anzeigen, wo der Anwender mit der Mitteilung nichts anzufangen weiß.

Den Fehlertext können Sie mit Hilfe der Eigenschaft *ErrorMessage* festlegen. Weitere Gestaltungsmöglichkeiten bieten sich mit den allgemeinen Eigenschaften wie *Font*, *ForeColor* etc. an.

BEISPIEL 40.41: *RequiredFieldValidator*-Control

[1] Wie bei MS nicht anders zu erwarten, natürlich der IE.

[2] Hiermit sollen Manipulationsmöglichkeiten auf der Clientseite ausgeschlossen werden.

Starten Sie die Anwendung, um sich von der Funktionstüchtigkeit zu überzeugen. Zunächst ist kein Unterschied festzustellen, versuchen Sie jedoch die Berechnung zu starten bzw. die Formulareingaben zum Server zu senden, wird die vordefinierte Fehlermeldung angezeigt:

Geben Sie nun einen Text in die Textbox ein und versuchen Sie erneut Ihr Glück, so verschwindet die Fehlermeldung von allein.

Natürlich vereinfacht die Tatsache, dass Eingabewerte nach einem erneuten Anzeigen der Seite bestehen bleiben, die Programmierung derartiger Prüfungen wesentlich. Wer möchte schon nach dem Ausfüllen von 10 oder 20 Feldern alles noch einmal eingeben, nur weil vielleicht ein einziger Wert falsch eingegeben wurde?

HINWEIS: Im Zusammenhang mit der Verwendung von Validator-Controls sollten Sie *AutoPostBack* besser auf *False* stellen!

Auf die Besonderheiten der verschiedenen Controls wird in den folgenden Abschnitten im Detail eingegangen.

40.4.4 RequiredFieldValidator

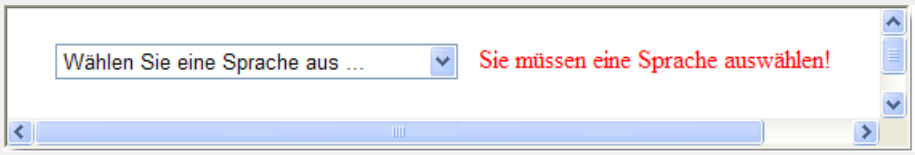
Diese Komponente haben Sie ja bereits im vorhergehenden Beispiel kennen gelernt. Die Aufgabe des *RequiredFieldValidators* besteht in der Prüfung, ob sich überhaupt ein Wert im zu überwachenden Control (z.B. *TextBox*) befindet. Ist dies nicht der Fall, wird der mit *ErrorMessage* festgelegte Text ausgegeben.

Doch was hat es mit der Eigenschaft *InitialValue* auf sich? Die Antwort findet sich beispielsweise bei einer *DropDownList*, die ja bereits beim Laden der Seite den ersten Wert anzeigt. Möchten Sie, dass der Nutzer einen anderen Wert auswählen muss, geben Sie den ersten Wert (Defaultwert) der *DropDownList* als *InitialValue* für den Validator vor.

BEISPIEL 40.42: Mit einer *DropDownList* soll eine Sprache ausgewählt werden. Zulässig sind nur die drei letzten Auswahlpunkte:

BEISPIEL 40.42: Mit einer *DropDownList* soll eine Sprache ausgewählt werden. Zulässig sind nur die drei letzten Auswahlpunkte:

Wählt der Nutzer keinen anderen Wert aus, tritt das *Validator*-Control in Aktion:



40.4.5 CompareValidator

Mit dem *CompareValidator* lassen sich bereits wesentlich komplexere Regeln aufstellen. Das Grundprinzip ist hier ein Vergleich mit konstanten Werten, einem anderen Feld oder mit einem Datentyp.

Wichtig sind die folgenden Eigenschaften:

Eigenschaft	Beschreibung
ControlToCompare	(optional) ein Control, mit dem der Vergleich erfolgen soll
ControlToValidate	das zu überwachende Control
Operator	der verwendete Vergleichsoperator
Type	(optional) ein vorgegebener Datentyp
ValueToCompare	(optional) ein konstanter Vergleichswert

Über die *Operator*-Eigenschaft können Sie einen der folgenden Werte auswählen:

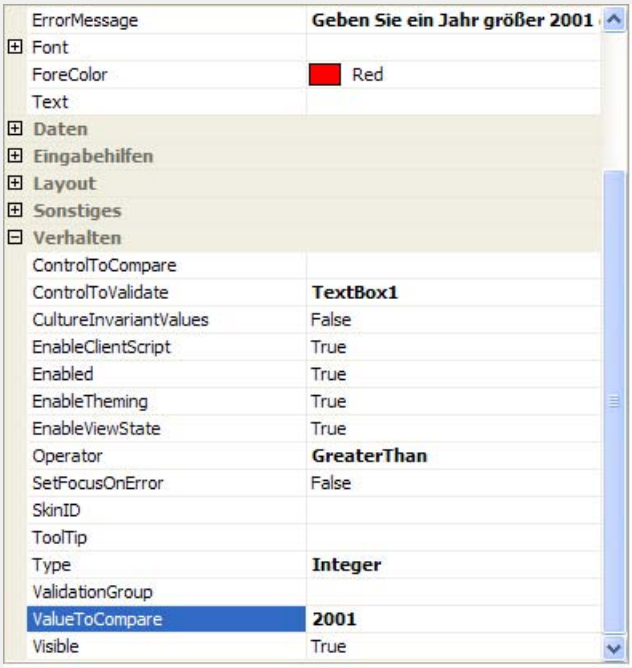
Wert	Beschreibung
Equal	gleich
GreaterThan	größer
GreaterThanEqual	größer gleich
LessThan	kleiner
LessThanEqual	kleiner gleich
NotEqual	ungleich
DataTypeCheck	nur Datentypvergleich (*Integer, String, Double, Currency, Date*)

BEISPIEL 40.43: Wir möchten die Eingabe einer Jahreszahl nach 2001 festlegen.

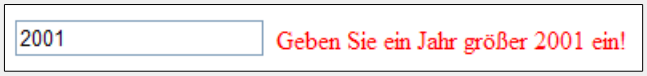
Realisierung: Platzieren Sie ein *CompareValidator*-Control neben dem Eingabefeld für die Jahreszahl und verknüpfen Sie die beiden Controls über die *ControlToValidate*-Eigenschaft. Nachfolgend legen Sie die Eigenschaft *Operator* auf *GreaterThan* fest. In der Eigenschaft *ValueToCompare* bringen Sie den Vergleichswert "2001" unter.

BEISPIEL 40.43: Wir möchten die Eingabe einer Jahreszahl nach 2001 festlegen.

Was bleibt, ist das Festlegen einer geeigneten Fehlermeldung mittels *ErrorMessage*:



Ein kurzer Test mit der Eingabe "2001" zeigt das gewünschte Ergebnis:

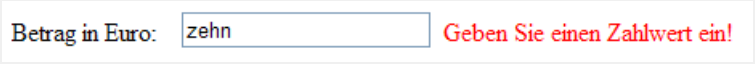


BEISPIEL 40.44: Bei der Eingabe im Feld "Betrag" soll es sich um einen Währungswert handeln.

Bisher erfolgt lediglich eine Prüfung, ob überhaupt etwas eingegeben wurde (ein Buchstabe genügt, um die Bedingung zu erfüllen).

Realisierung: Fügen Sie ein weiteres *CompareValidator*-Control ein und verknüpfen Sie dieses mit der gewünschten Textbox. Die Eigenschaft *Operator* legen Sie auf *DataTypeCheck* fest. Mit *Type* wählen Sie den Datentyp "Currency" aus.

Der Test mit einer Falscheingabe zeigt das gewünschte Ergebnis (die Bedingung "Eingabe erforderlich" ist zwar erfüllt, aber der Datentyp stimmt nicht):



Wie Sie sehen, ist es kein Problem, einem Eingabe-Control mehrere Validator-Controls zuzuordnen. Lediglich die Platzierung der Controls auf dem Web-Form dürfte im Laufe der Zeit recht unübersichtlich werden. Abhilfe schafft das *ValidationSummary*-Control, auf das wir im Abschnitt 40.4.9 näher eingehen werden.

40.4.6 RangeValidator

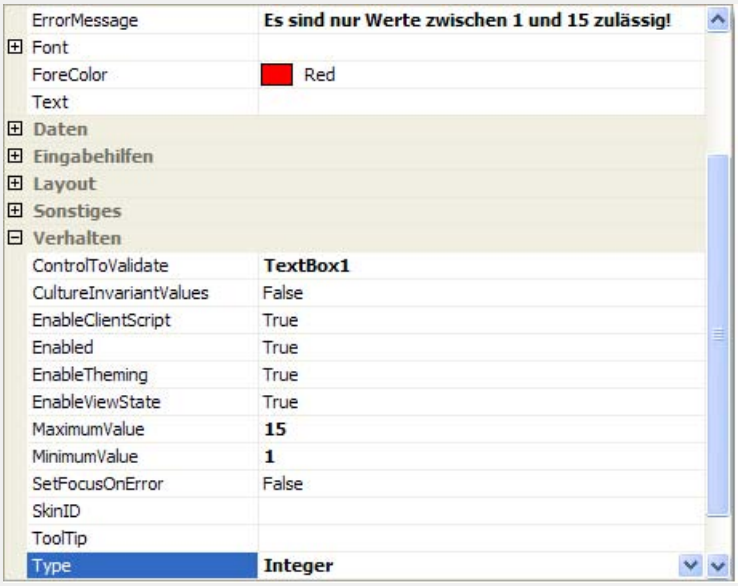
Mit dem *RangeValidator* kann, wie es der Name schon vermuten lässt, ein Eingabewert auf Einhaltung eines bestimmten Wertebereichs geprüft werden.

Legen Sie zunächst den Datentyp mit der *Type*-Eigenschaft fest. Dies ist wichtig, da es natürlich einen Unterschied zwischen *String*- und *Integer*-Bereichen gibt.

Den eigentlichen Bereich stellen Sie mit den Eigenschaften *MinimumValue* und *MaximumValue* ein. Der Eingabewert muss dann größer gleich dem Minimum und kleiner gleich dem Maximum sein.

BEISPIEL 40.45: Eine Anwendung soll lediglich die Eingabe von Laufzeiten zwischen 1 und 15 Jahren zulassen.

Wie Sie die Eigenschaften festlegen müssen, zeigt die folgende Abbildung:

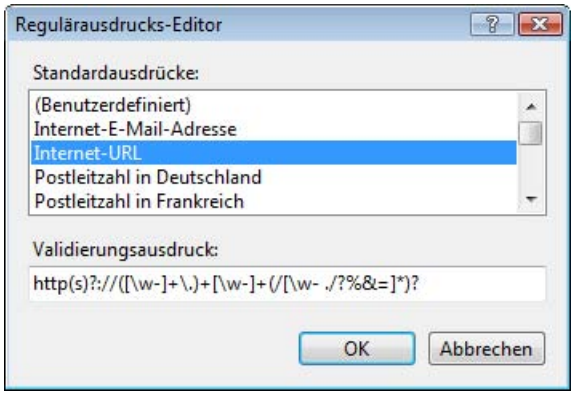


40.4.7 RegularExpressionValidator

Wem die bisherigen Möglichkeiten der Prüfung von Eingabewerten nicht ausreichen, der findet sicher bei den regulären Ausdrücken das Gewünschte. Mit Hilfe einer Beschreibungssprache (Steuerzeichen, Platzhalter etc.) können quasi Masken für Eingabewerte erstellt werden.

Prominente Beispiele sind Ihnen sicher schon auf mehreren Webseiten begegnet:

- E-Mail-Adressen
- Web-Adressen
- Kreditkartennummern
- Telefonnummern (mit nationaler bzw. internationaler Vorwahl)
- Postleitzahlen …



Der für den das alles "böhmische Dörfer" sind, der sollte sich einmal die Beispiele ansehen, die über den *ValidationExpression*-Eigenschaftseditor bereitgestellt werden.

HINWEIS: Mehr zu den Hintergründen und der verwendeten Syntax finden Sie in der MSDN-Library unter dem Stichwort ".NET Framework Regular Expressions".

Die weiteren Eigenschaften des Controls entsprechen denen der anderen Validator-Controls.

40.4.8 CustomValidator

Fehlte Ihnen bisher die Möglichkeit, selbst Hand anzulegen? Wenn ja, sind Sie genau hier richtig.

Der *CustomValidator* bietet Ihnen zwei grundsätzliche Prüfungsmöglichkeiten, die Sie durch eigene Routinen realisieren können:

- serverseitige Validierung
- clientseitige Validierung

Während bei der ersten Variante ein Ereignis auf dem Server ausgelöst wird und Sie wie gewohnt mit C# programmieren können, lässt die zweite Variante Erinnerungen an alte Zeiten aufkommen. Sie erzeugen ein clientseitiges Skript (VBScript, JavaScript) mit allen Unwägbarkeiten, die damit zusammenhängen.

Clientseitige Validierung

Ihre Client-Validierungsfunktion hat einige Grundanforderungen zu erfüllen. So hat die Funktion zwei Parameter (*source* und *args*) zur Verfügung zu stellen. Zusätzlich muss ein boolescher Rückgabewert realisiert werden.

BEISPIEL 40.46: Testen, ob ein Wert kleiner 10 eingegeben wurde

HTML

Die Konfiguration des *CustomValidator*-Controls:

```
...
  <asp:CustomValidator ID="CustomValidator1" runat="server"
                       ClientValidationFunction="AllesOk"
                       ErrorMessage="Zahl kleiner 10 eingeben!"
                       ControlToValidate="TextBox1"></asp:CustomValidator>
```

JavaScript

Die JavaScript-Funktion am Ende der ASPX-Seite:

```
...
</body>
<script language=javascript>
<!--
    function AllesOk(source, args)
    {
      args.IsValid = (parseInt(args.Value) < 10);
    }
// -->
</script>
</html>
```

HINWEIS: Ganz nebenbei könnten Sie natürlich per Client-Script auch Dialogboxen einblenden.

Serverseitige Validierung

Wie bei einer ereignisgesteuerten Programmierung wohl nicht anders zu erwarten ist, müssen Sie folgende Ereignisprozedur zur Verfügung stellen:

```
protected void CustomValidator1_ServerValidate(object source, ServerValidateEventArgs args)
{
  ...
}
```

Über den Parameter *args* haben Sie zum einen Zugriff auf den Inhalt des zu überwachenden Controls (*args.Value*-Eigenschaft), zum anderen bestimmen Sie mit *args.IsValid,* ob die Prüfung erfolgreich war oder auch nicht.

BEISPIEL 40.47: Serverseitige Validierung

C#

Wir möchten ein Eingabefeld prüfen, es soll sich um einen numerischen Wert handeln, der nicht größer als 10,4 und nicht kleiner als 0,5 ist.

BEISPIEL 40.47: Serverseitige Validierung

Realisierung: Platzieren Sie ein *CustomValidator* neben dem Eingabefeld für den Zinssatz und verbinden Sie die beiden Controls miteinander über die *ControlToValidate*-Eigenschaft. Nachfolgend klicken Sie doppelt auf das *CustomValidator*-Control, um die Ereignisprozedur zu erstellen:

```
protected void CustomValidator1_ServerValidate(object source, ServerValidateEventArgs args)
{
 Single n;
```

Zunächst prüfen wir den Datentyp:

```
  try
 {
   n = Convert.ToSingle(args.Value);
```

Danach den Wertebereich:

```
    args.IsValid = (n >= 0.5)&&(n <= 10.4);
 }
 catch (Exception)
 {
   args.IsValid = false;
  }
}
```

HINWEIS: Doch Achtung: Testen Sie das Beispiel wie oben angegeben, passiert nichts bzw. es wird ein Fehler auftreten.

Im Gegensatz zu den anderen Validator-Controls, die bereits auf dem Client die entsprechenden Prüfungen vornehmen, erfolgt die Prüfung auf dem Server. Doch leider funktioniert die *AutoPostBack*-Funktion in diesem Fall nicht wie vorgesehen, die Ereignismethode wird nicht ausgelöst.

Erst mit dem Klick auf den Button wird auch das Ereignis ausgelöst, was allerdings etwas spät erfolgt, da wir nun bereits das zweite Formular anzeigen wollen. Deshalb müssen Sie eine weitere Prüfung beim *Button_Click* vornehmen:

```
    protected void Page_Load(object sender, EventArgs e)
    {
       if (Page.IsValid)
       {
...
       }
    }
```

Jetzt klappt es auch mit der Prüfung, der Fehlertext wird korrekt angezeigt.

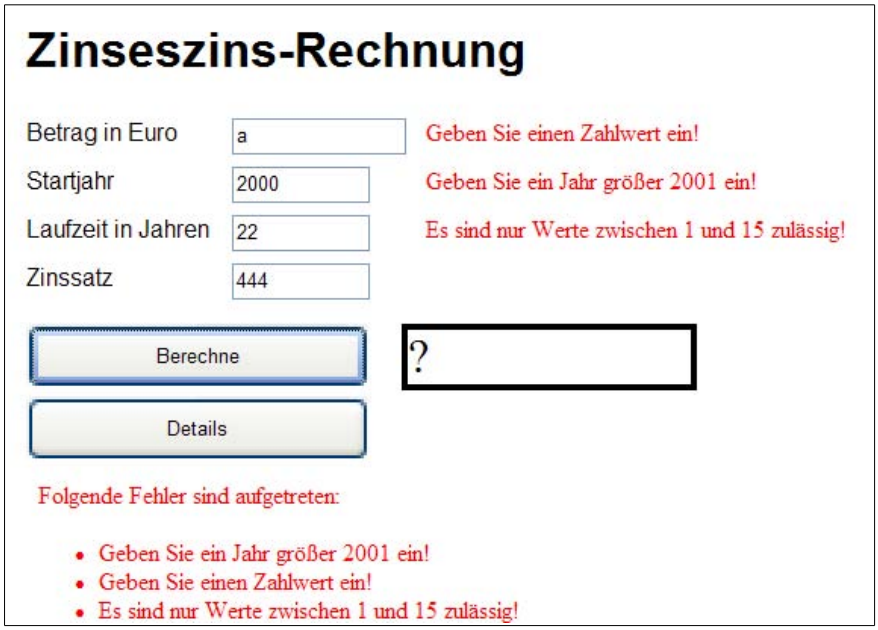
Gleichzeitig haben Sie auch einen weiteren Weg kennen gelernt, wie Sie den Fehlerstatus in Ereignisprozeduren abfragen können.

HINWEIS: Ein wichtiges Anwendungsgebiet für das *CustomValidator*-Control dürften Abfragen in Datenbanken oder Verzeichnissen sein (z.B. Passwort/Name etc.).

40.4.9 ValidationSummary

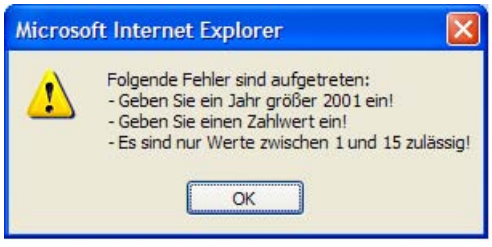
Wem die bisherigen Varianten etwas zu unübersichtlich waren und wer an einer zentralen Anzeige der Eingabefehler interessiert ist, der sollte sich das *ValidationSummary*-Control näher ansehen.

Mit diesem Control werden selbst keinerlei Regeln definiert, Sie müssen das Control noch nicht einmal an die anderen Controls binden. *ValidationSummary* zeigt lediglich die Fehlermeldungen der anderen Validation-Controls in einer übersichtlichen Form an, wie folgendes Beispiel zeigt:



Mit der Eigenschaft *DisplayMode* steuern Sie, wie die Fehlerliste ausgegeben werden soll. Möglich sind die Werte *List*, *BulletList*, *SingleParagraph*. Weiterhin dürfte *HeaderText* von Interesse sein, damit legen Sie einen einführenden Text vor den eigentlichen Fehlermeldungen fest.

ShowMessageBox ermöglicht Ihnen die zusätzliche clientseitige Anzeige eines Meldungsfensters:



Reicht Ihnen obige MessageBox, können Sie mit *ShowSummary=False* die eigentliche Anzeige der Fehler verhindern.

40.4.10 Weitere Möglichkeiten

Validierung zeitweise verhindern

Leider hat die konsequente Validierung auch ihre Schattenseiten. Stellen Sie sich den Fall vor, bei dem die Eingabe in ein Formular vom Nutzer abgebrochen werden soll. In Windows-Anwendungen fällt Ihnen jetzt sicher spontan der obligate Abbruch- oder Cancel-Button ein. Doch in Webanwendungen stehen Sie vor einem Problem: Jeder Klick auf einen Button hat auch eine Verbindung zum Server zur Folge, und dies wiederum führt unvermeidlich zu einer Fehlerprüfung, die zu diesem Zeitpunkt überflüssig ist.

Aus diesem Grund verfügen alle Controls, die eine automatische Verbindung zum Server auslösen, über die Eigenschaft *CausesValidation* (Default=*True*), mit der die Fehlerprüfung gezielt ein- bzw. ausgeschaltet werden kann.

ValidationGroup bilden

Es besteht die Möglichkeit, mehrere Validation-Controls zu einer logischen Gruppe zusammenzufassen. Es ist naheliegend, dass dabei die Eigenschaft *ValidationGroup* eine zentrale Rolle spielt. Gemeinsames Ausgabe-Control ist ein *ValidationSummary*-Steuerelement.

Eingabefokus setzen

Mit der Eigenschaft *SetFocusOnError* können Sie erreichen, dass nach einer fehlerhaften Eingabe der Fokus auf das betreffende Steuerelement gesetzt wird.

40.5 Praxisbeispiele

40.5.1 Themes und Skins verstehen

Sie möchten dem Nutzer Ihrer Website etwas Gutes tun und eine Auswahl verschiedener Seitenlayouts (Themes) bereitstellen. Mit dem Klick auf die jeweilige Schaltfläche soll das Theme sofort auf die aktuelle Seite angewendet werden.

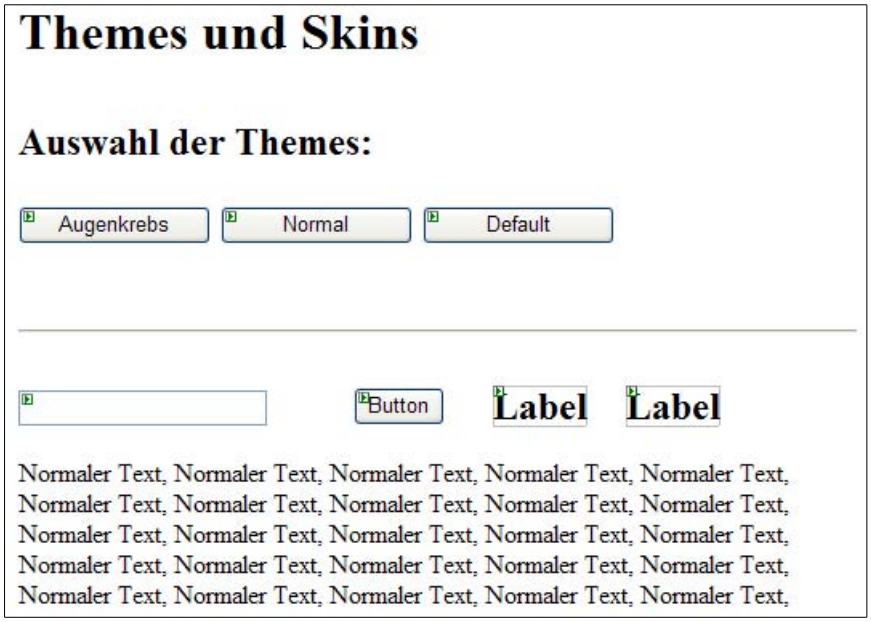
Für unser Beispiel belassen wir es bei drei Auswahlmöglichkeiten:

- "Augenkrebs" – ein grauenhaft buntes Layout, das junge Leute ansprechen soll,
- "Normal" – für den Rest der Bevölkerung
- "Default" – zurücksetzen auf die Web-Standardeinstellungen

Neben dem Layout von Steuerelementen soll auch per *Cascading Style Sheet* auf die normalen HTML-Elemente (H1, H2, Body ...) Einfluss genommen werden.

Oberfläche

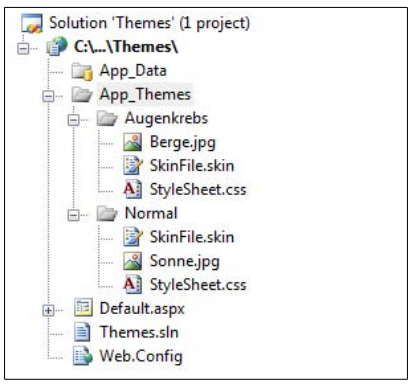
Erstellen Sie ein neues Webprojekt (ASP.NET Anwendung) und entwerfen Sie folgende Oberfläche:



Projekt

Klicken Sie im Projektmappen-Explorer auf das Projekt und öffnen Sie das Kontextmenü. Über den Menüpunkt *Verzeichnis hinzufügen* fügen Sie ein *Theme Verzeichnis* hinzu. Der Name des Verzeichnisses ist gleichzeitig der Theme-Name. In dieses Verzeichnis fügen Sie über *Neues Element hinzufügen* ein Skin-File (*SkinFile.skin*) und ein Style Sheet-File (*StyleSheet.css*) hinzu. Wiederholen Sie den Vorgang für alle gewünschten Themes.

Das Projekt nach dem Einfügen:



Skin bearbeiten

Öffnen Sie nun eine der Skin-Dateien (Theme *Augenkrebs*) und fügen Sie die beiden folgenden Einträge hinzu:

```
<asp:TextBox runat="server" Style="font-weight: bold; color: #ffff00;
            border-top-style: none; border-right-style: none; border-left-style: none;
            background-color: #66ff33; border-bottom-style: none"></asp:TextBox>

<asp:Button runat="server" Style="border-right: #ff00cc thick solid;
            border-top: #ff00cc thick solid; font-weight: bold; font-size: 12pt;
            border-left: #ff00cc thick solid; color: #ff0000;border-bottom: #ff00cc
            thick solid; font-family: arial; background-color: #00ff00"/>
```

Wie Sie sehen, muss für jedes gewünschte Steuerelement eine komplette Beschreibung abgelegt werden. Doch keine Bange, Sie müssen sich die einzelnen Einträge nicht mühsam aus der Hilfe zusammenlesen. Es genügt, wenn Sie auf einem Webformular das jeweilige Steuerelement so bearbeiten, wie es Ihnen gefällt, der erzeugte Quellcode (HTML-Source-Ansicht) muss nur noch von steuerelementspezifischen Attributen (*Name*, *Text*) gesäubert und in die Skin-Datei eingefügt werden.

BEISPIEL 40.48: Fette Markierungen entfernen

HTML

```
asp:Button ID="Button5" runat="server" Style="border-right: silver thick dotted;
                border-top: silver thick dotted; border-left: silver thick dotted;
                color: lime; border-bottom: silver thick dotted; background-color: green"
                Text="Button" />
```

Wiederholen Sie den Vorgang für alle Themes bzw. deren Skin-Dateien.

HINWEIS: Wollen Sie für ein Steuerelement mehrere Layouts in einem Skin ablegen, fügen Sie ein Attribut *skinid* hinzu. Über dieses entscheidet das spätere Steuerelement, welches Layout gewünscht ist.

BEISPIEL 40.49: Zwei Labeltypen

HTML

```
<asp:Label skinid="Variante1" runat="server" Style="background-color: gray"></asp:Label>
<asp:Label skinid="Variante2" runat="server" Style="background-color: red"></asp:Label>
```

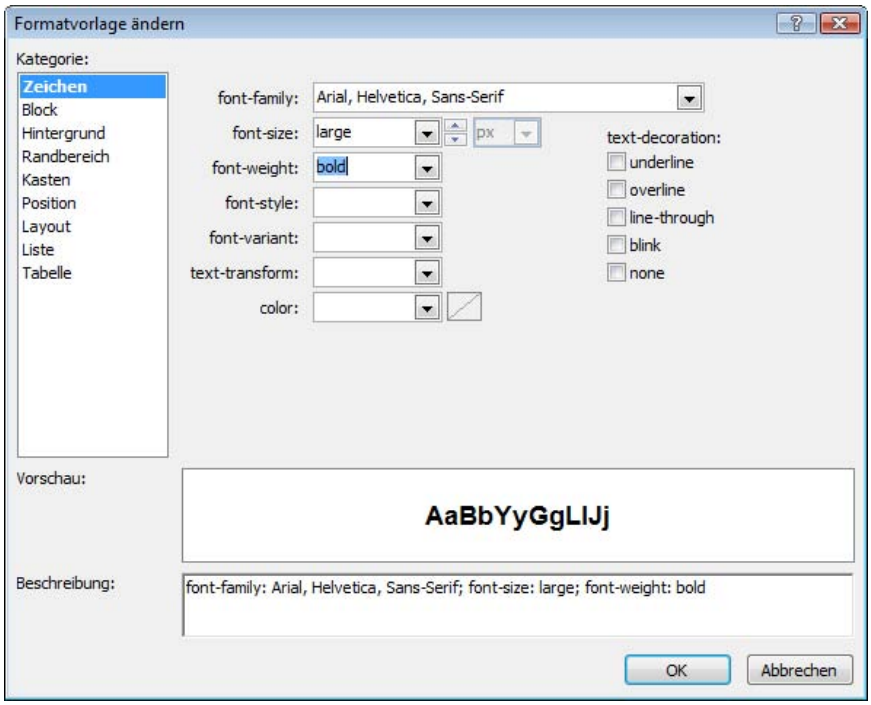
Style Sheet bearbeiten

An dieser Stelle wollen wir nicht über die Bedeutung und die Möglichkeiten von Style Sheets diskutieren, übernehmen Sie einfach das folgende einfache Style Sheet:

```
h1 { font-size: 20pt;margin-bottom: 18pt; color: #0033ff; font-family: Arial;}

h2 { font-size: 18pt; margin-bottom: 18pt;
 color: #00ffcc;font-family: Arial; }
```

```
h3 { font-size: 12pt; margin-top: 20pt; color: #33ff33; font-family: Arial;
 font-weight: bold; }

h4 {   font-size:12pt; }

body { font-size: 10pt; margin-bottom: 3pt;color: #ff6600;font-family: Arial;
 background-color: greenyellow; }
```

HINWEIS: Nutzen Sie ausgiebig den Stil-Generator, der eine einfache Bearbeitung der Einträge ermöglicht.



Auch hier müssen/können Sie für jedes Theme ein eigenes Style Sheet anlegen.

Quelltext

Die Auswahl des Theme wollen wir dem Anwender überlassen. Mit dem Klick auf einen der drei Auswahlbuttons – wir hätten auch ein anderes Steuerelement verwenden können – soll das neue Theme dargestellt werden.

Nichts einfacher als das, doch da tut sich ein kleines Problem auf: Die Zuweisung des Theme per Code muss **im oder vor** dem Aufruf des Ereignisses *Page_PreInit* erfolgen. Zu diesem Zeitpunkt ist aber weder eine Ereignismethode abgearbeitet, noch können Sie auf den Inhalt von Steuerelementen zugreifen. Abhilfe schafft der gute alte *QueryString*, der per URL übergeben wird.

Statt Ereignisprozeduren zu schreiben, müssen wir also die Eigenschaft *PostBackUrl* der drei Buttons setzen:

```
Button1.PostBackUrl = ~/Default.aspx?Theme=Augenkrebs
Button2.PostBackUrl = ~/Default.aspx?Theme=Normal
Button3.PostBackUrl = ~/Default.aspx?Theme=Default
```

Die Auswertung können wir dann im Ereignis *Page_PreInit* vornehmen:

```
void Page_PreInit(object sender, EventArgs e)
{
  switch (Request.QueryString["Theme"])
  {
    case "": break;
    case "Default":
      Session["Theme"] = "";
      Page.Theme = (string) Session["Theme"];
      break;
    default:
      Session["Theme"] = Request.QueryString["Theme"];
      Page.Theme = (string) Session["Theme"];
      break;
  }
}
```

Warum das zusätzliche Abspeichern in einer *Session*-Variablen?

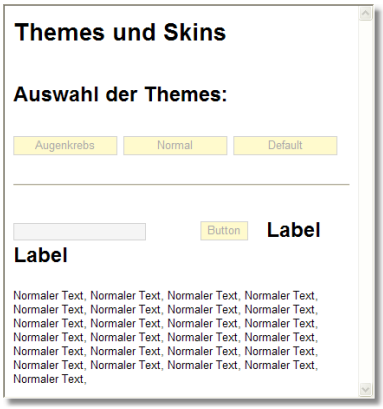
Die Antwort ist schnell gefunden, wenn Sie mehr als eine Seite in Ihrem Projekt haben, braucht sich das jeweilige Formular nur an die *Session*-Variable zu halten.

Test

Starten Sie die Webanwendung und klicken Sie auf die drei oberen Buttons. Das (mögliche) Ergebnis (Hoffentlich verderben Sie sich nicht die Augen!):

Augenkrebs:

Normal:



HINWEIS: Natürlich können Sie auch noch Bilder etc. mit den Themes austauschen, für einen ersten Einblick dürften unsere Ausführungen aber gereicht haben.

40.5.2 Masterpages verwenden

Nach der doch sehr theoretischen Einführung in diesem Kapitel wollen wir Ihnen ein praktisches Beispiel nicht vorenthalten. Ziel unserer Bemühungen ist eine recht einfache Anwendung, in der drei Seiten mit gleichem Layout (einheitlicher Master) verwaltet werden sollen. Alle Seiten sollen eine Navigationsmöglichkeit aufweisen.

Oberfläche (Masterpage)

Erstellen Sie ein neues ASP.NET-Projekt und löschen Sie die automatisch erstellte Startseite *default.aspx*.

Erzeugen Sie nun über *Website|Neues Element hinzufügen* eine Masterpage mit folgendem Layout:

HINWEIS: Die Steuerelemente sind in einer HTML-Table enthalten. Diese ermöglicht uns die sinnvolle Positionierung der Buttons am linken Rand.

In der *ListBox* am linken Rand werden wir die einzelnen Seitenaufrufe protokollieren.

Die Navigation zu den einzelnen Seiten werden wir nicht per Ereignis, sondern über die Eigenschaft *PostBackUrl* der Buttons realisieren.

```
Button1.PostBackUrl=~/Default.aspx
Button2.PostBackUrl=~/Default2.aspx
Button3.PostBackUrl=~/Default3.aspx
```

Ob Sie, wie im vorliegenden Beispiel, noch eine Hintergrundgrafik zuweisen, bleibt Ihnen überlassen.

HINWEIS: Soll der Content-Bereich einfarbig markiert sein, genügt es, wenn Sie die entsprechenden Tabellenzellen formatieren.

Damit sind alle Vorarbeiten abgeschlossen und wir wenden uns den Contentpages zu.

Oberfläche (Contentpage)

Erstellen Sie die Contentpages wie "normale" WebForms, beachten Sie jedoch, dass Sie die Option "Masterpage auswählen" markieren. Weisen Sie im folgenden Schritt die bereits erzeugte Masterpage zu.

Die neue Seite dürfte dann etwa den folgenden Anblick bieten:

Fügen Sie etwas in die Seite ein, damit Sie die einzelnen Seiten später noch unterscheiden können. Wiederholen Sie das Ganze noch für zwei weitere Seiten, um auch einen echten Umschalteffekt zu sehen.

Quelltext

Eigentlich ist keinerlei Quelltext für die Funktion der Website notwendig, wir wollen jedoch noch eine kleine Protokollfunktion implementieren, die Ihnen die Arbeitsweise der Masterpages verdeutlichen soll.

In der Masterpage:

```
protected void Page_Load(object sender, EventArgs e)
 {
```

```
    ListBox1.Items.Add("Masterpage");
   }
```

In den Contentpages:

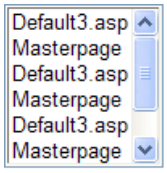
```
   protected void Page_Load(object sender, EventArgs e)
   {
    (Master.FindControl("ListBox1") as ListBox).Items.Add("Default.asp");
   }
```

Test

Starten Sie Ihre Website und wechseln Sie über die Buttonliste zwischen den einzelnen Seiten. Überlegen Sie sich, was für Einträge Sie in der *ListBox* erwarten.

Tja, warum werden immer nur zwei Einträge in der *ListBox* ausgegeben? Die Antwort ist schnell gefunden, wenn Sie sich in Erinnerung rufen, dass es sich bei der Masterpage und dem enthaltenen *ContentPlaceHolder* nicht um einen statischen, übergeordneten Frame handelt, sondern quasi um eine Schablone für die einzelnen Contentseiten.

Jede Seite hat ihre eigene *ListBox,* auch wenn diese durch den Master gekapselt wird. Der Seitenstatus geht bei einem Wechsel zwischen den Seiten verloren, nicht jedoch bei einem erneuten Aufruf derselben Seite:



40.5.3 Webparts verwenden

In diesem Beispiel wollen wir Ihnen die wichtigsten Grundzüge der Webpart-Programmierung anhand einer simplen Aufgabenstellung demonstrieren.

In einer dreispaltigen Webseite soll neben einem Kalender und einem Image ein einfaches User Control angezeigt werden. Die drei genannten Steuerelemente sollen sowohl in der Reihenfolge als auch in der Position (linke Spalte/rechte Spalte) zur Laufzeit frei verändert werden können.

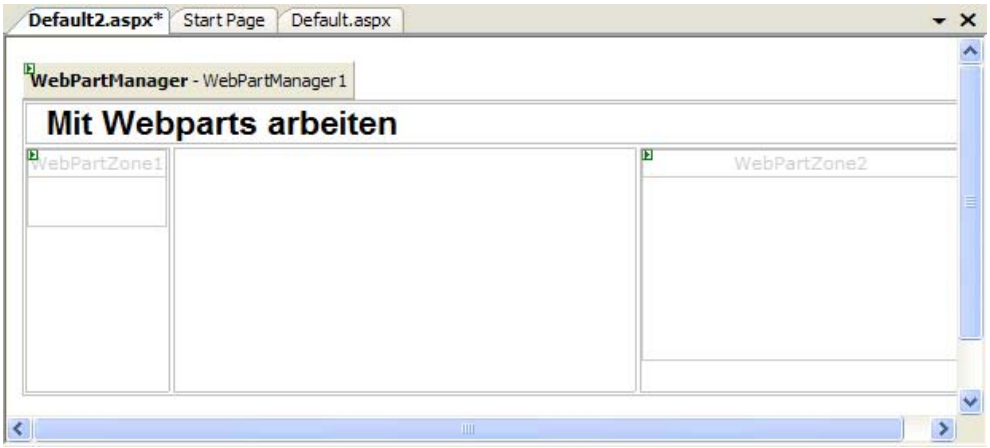
Vorbereiten der Portalseite

Erstellen Sie wie gewohnt ein WebForm-Projekt und fügen Sie in die Default-Seite ein *WebPartManager*-Steuerelement ein.

HINWEIS: Dieses Steuerelement muss das Erste auf der Seite sein!

Darauf aufbauend müssen Sie sich jetzt Gedanken um die Grundstruktur Ihrer Webseite machen, d.h. die Einteilung in einzelne, variable und statische Bereiche. Diese entsprechen den *WebPartZone*-Steuerelementen, die Sie am besten per HTML-Table positionieren und formatieren.

Für unser Beispiel erzeugen wir folgende Tabelle mit zwei *WebPartZone*-Steuerelementen (links/rechts):



In diese WebPartZones, die zur Laufzeit nicht direkt sichtbar sind (lediglich Platzhalter), können Sie Server-Steuerelemente (Buttons, Kalender etc.) oder auch User Controls einfügen.

User Control programmieren

Da wir im Weiteren auch auf ein User Control zurückgreifen wollen, kümmern wir uns zunächst um dessen Programmierung.

Erzeugen Sie ein neues *Web User Control* über den Menüpunkt *Website|Neues Element hinzufügen*. Diese Control (*WebUserControl.ascx*) können Sie fast wie eine normale WebForm-Seite programmieren, d.h., Sie platzieren Steuerelemente und fügen Ereigniscode hinzu.

HINWEIS: Beachten Sie jedoch, dass Größe und Position des User Controls von der später übergeordneten Webseite bestimmt wird. Enthaltene Steuerelemente sollten sich also an die Größe automatisch anpassen.

In unserem Beispiel wollen wir lediglich Einträge einer *TextBox* in eine *ListBox* einfügen:

Die erforderliche Programmlogik ist mit einer Zeile realisiert:

```
protected void Button1_Click(object sender, EventArgs e)
{
  ListBox1.Items.Add(TextBox1.Text);
}
```

Speichern Sie Ihr neues User Control ab.

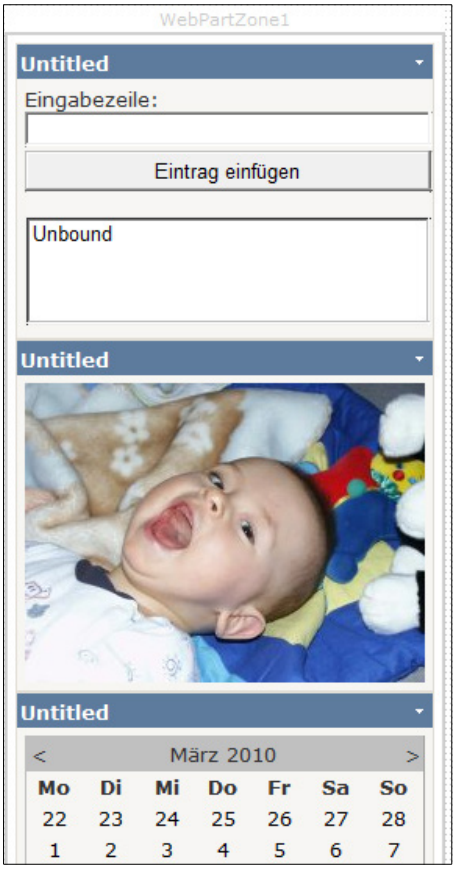
Inhalte für die Webzones definieren

Beispielhaft für andere Steuerelemente/Inhalte fügen wir

- unser neu erzeugte User Control (*WebUserControl.ascx*),
- ein *Image*
- und ein *Calendar*-Steuerelement

in *WebPartZone1* ein (siehe folgende Abbildung).

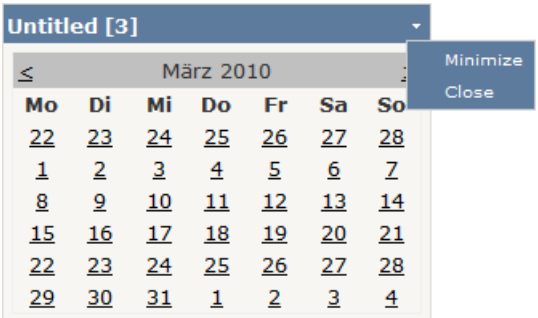
Wie Sie sehen, werden den einzelnen Steuerelementen automatisch Rahmen mit Kopfzeile hinzugefügt. Diese Rahmen bzw. die übergeordnete *WebPartZone* (diese vererbt quasi das Aussehen und Verhalten) lassen sich über Unmengen von Eigenschaften parametrieren. Wir belassen es dabei, über *AutoFormat* eine halbwegs ansprechende Formatierung auszuwählen.



Damit sind zunächst die Inhalte der Webseite festgelegt, und wir müssen uns um die Bearbeitung zur Laufzeit kümmern.

Programmieren des Werkzeugkastens

So wie die Seite aktuell programmiert ist, kann der Anwender zur Laufzeit lediglich die einzelnen Webparts, d h. unsere Steuerelemente, minimieren:



Ein Verschieben oder Konfigurieren ist nicht möglich, da sich die Seite lediglich im *BrowseDisplayMode* befindet. Soll etwas mehr Funktionalität zur Verfügung stehen, müssen Sie den *DisplayMode* der *WebPartManagers* ändern.

Wir beschränken uns auf den Modus *EditDisplayMode*, den wir mit einem zusätzlichen Button (*Editieren*) aktivieren:

```
protected void Button1_Click(object sender, EventArgs e)
{
  WebPartManager1.DisplayMode = WebPartManager.EditDisplayMode;
}
```

Damit können wir uns einem ersten Test zuwenden.

Starten Sie die Anwendung und klicken Sie auf den Button "Editieren", um in den *EditDisplayMode* zu schalten.

Automatisch werden die bisher nicht sichtbaren WebPartZonen eingeblendet und Sie können per Drag & Drop die Steuerelemente zwischen den Zonen verschieben bzw. auch die Reihenfolge der Elemente innerhalb einer Zone ändern:

Soll es etwas mehr Funktionalität sein (Layout-Änderungen) müssen wir noch etwas Hand anlegen.

Fügen Sie in die rechte Spalte der HTML-Table eine *EditorZone* ein. In diese (und nur in diese) können Sie Steuerelemente der Typen *AppearanceEditorPart* und *LayoutEditorPart* einfügen.

Erst jetzt steht über die Kopfzeile der Webparts zur Laufzeit der Menüpunkt *Edit* zur Verfügung. Klicken Sie darauf, wird die *EditorZone* mit den enthaltenen Steuerelementen angezeigt:

Hier können Sie den Titel der Webparts, die Ausrichtung und Größe etc. beeinflussen. Alle Änderungen werden über den aktuell gewählten Provider in der Projekt-Datenbank (Tabelle *aspnet_PersonalizationPerUser*) gespeichert und stehen bei einem erneuten Aufruf der Seite dem Nutzer wieder zur Verfügung.

Dass dieses System erst in Kombination mit dem Login, d h. der Zuordnung von Usern, einen Sinn macht, dürfte einleuchten. So wie das Projekt jetzt aufgebaut ist, wirken sich Änderungen eines Nutzers auf alle Anwender aus (Standard-Anmeldung).

Kapitel 41

Datenbindung unter ASP.NET

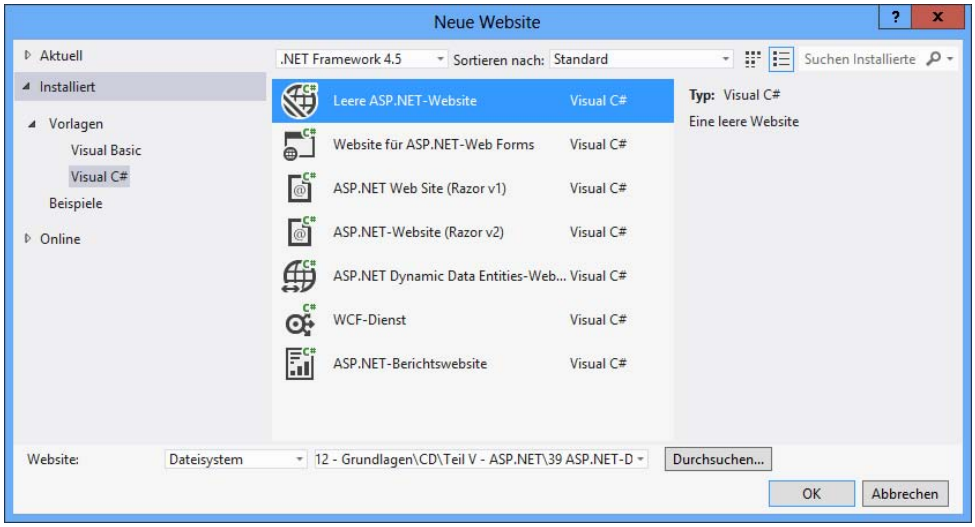
Eine der zentralen Aufgaben von ASP.NET ist das Darstellen und Editieren von Daten in verteilten Anwendungen auf der Basis von ADO.NET. Basierend auf dem Konzept der *DataSource*-Steuerelemente können Sie in kürzester Zeit relative komplexe Anwendungen erstellen, die teilweise sogar ganz ohne eigenen C#-Code auskommen.

Ein kurzes Einführungsbeispiel zeigt die Vorgehensweise, bevor wir uns mit den grundlegenden Details beschäftigen.

41.1 Einstiegsbeispiel

41.1.1 Erstellen der ASP.NET-Website

Öffnen Sie eine neue *Leere ASP.NET-Website* (*Datei/Neu/Website...*) und belassen Sie es bei der Standardeinstellungen für Speicherort (*Dateisystem*) und Pfad.

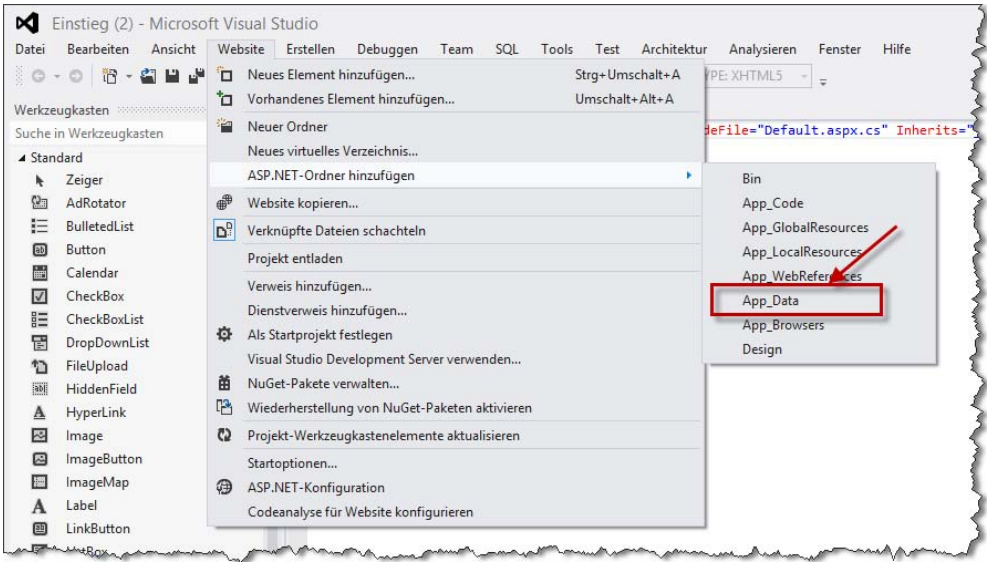


Nach dem Klick auf *OK* vergeht eine kleine Weile, bis die Web-Entwicklungsumgebung von Visual Studio erscheint.

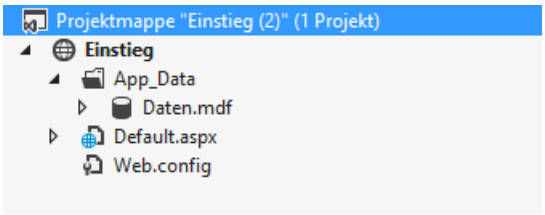
Fügen Sie jetzt zunächst eine neue Seite *Default.aspx* hinzu (*Hinzufügen|Web Form*), bevor wir uns um die Datenbank kümmern.

Datenbank hinzufügen

Handelt es sich bei Ihrer Datenbank um eine lokale Datenbank, wie zum Beispiel eine Access MDB oder eine SQL Server MDF-Datei, benötigen Sie ein vordefiniertes Verzeichnis *App_Data*, in dem die Datei abgelegt wird. Erstellen Sie dieses über den Menüpunkt *Website|ASP.NET-Ordner hinzufügen|App_Data*:



Nachfolgend können Sie die Datenbankdatei (in unserem Fall *Daten.mdf*) per Drag & Drop aus dem Windows-Explorer in das neue Verzeichnis kopieren:



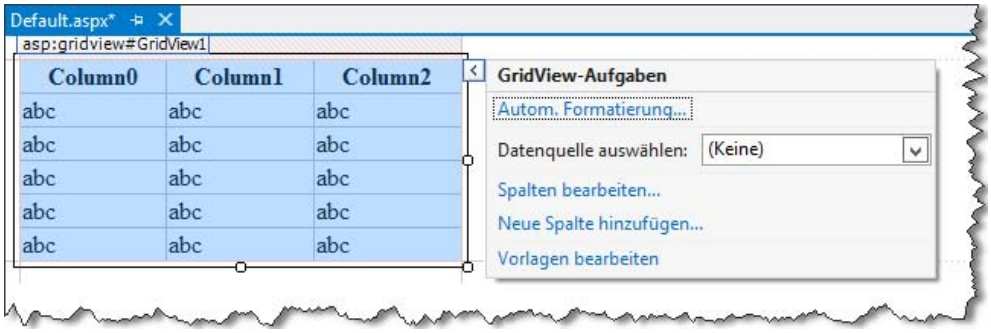
HINWEIS: Mit der Einführung von ASP.NET 4.5 verschwindet auch die *AccessDataSource* aus der Liste der verfügbaren Datenquellen. Microsoft rät grundsätzlich von der Verwendung von Access-Datenbanken ab.

GridView hinzufügen

Öffnen Sie die Datei *Default.aspx* und wechseln Sie in die Entwurfsansicht. Ziehen Sie ein *GridView* aus dem Werkzeugkasten (Kategorie *Daten*) auf die Oberfläche des Designers.

Da das *GridView* standardmäßig in einem recht spartanischen Outfit erscheint, empfehlen sich einige kosmetische Korrekturen, wie zum Beispiel das Ändern der Eigenschaften *BorderStyle, Font.Size, AlternatingRowStyle.BackColor*, *HeaderStyle.BackColor*, *Caption* ...

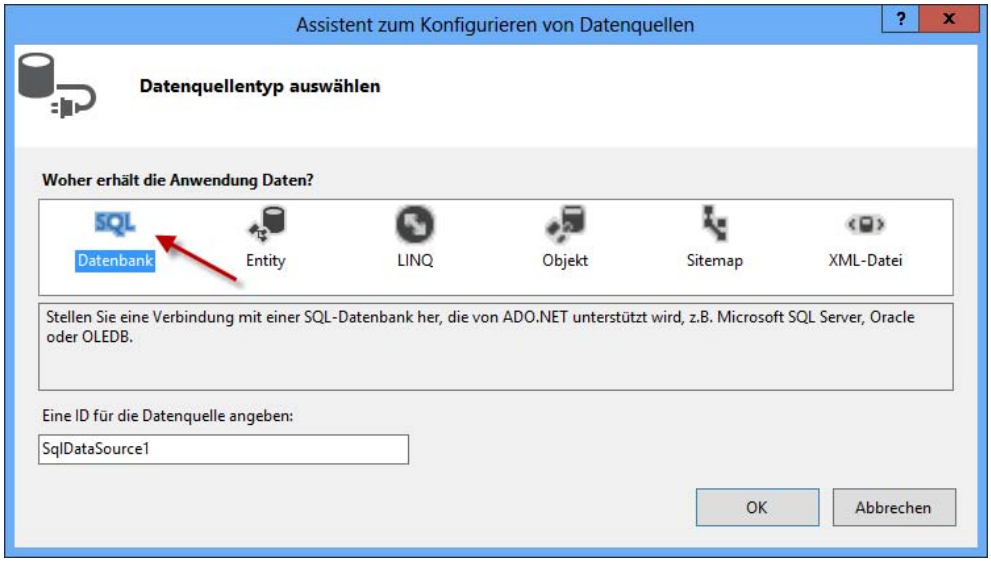
Wir nutzen nun die vom *GridView* angebotene Möglichkeit, eine Datenquelle per Aufgaben-Menü hinzuzufügen/zu erstellen:



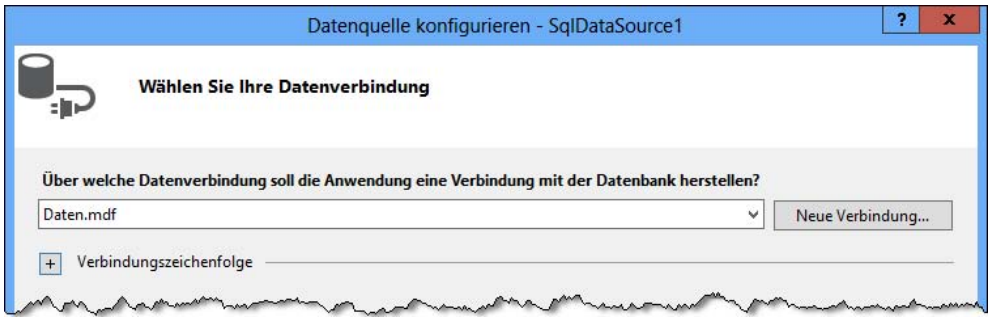
Klicken Sie dazu auf die Datenquellenauswahl und den Eintrag *<Neue Datenquelle>*.

GridView an Datenquelle anbinden

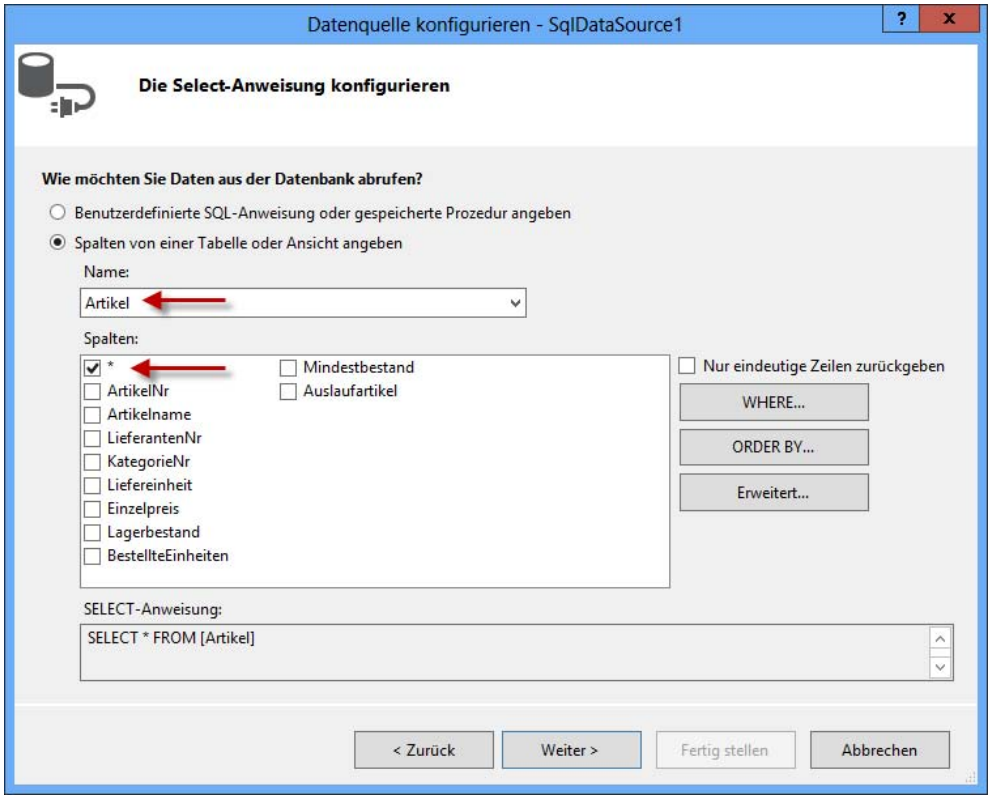
Im folgenden Dialog entscheiden wir uns für die SQL-Datenbank:



Wir wählen im folgenden Schritt unsere bereits vorhandene Datenbank *Nordwind.mdb* im Verzeichnis *App_Data* aus:

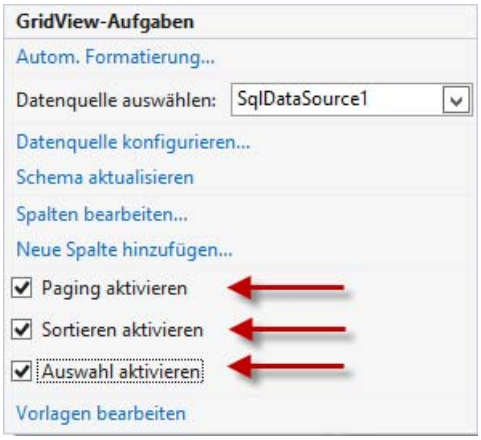


Speichern Sie nachfolgend die Verbindungszeichenfolge ab und wählen Sie eine Tabelle und deren gewünschte Spalten aus:



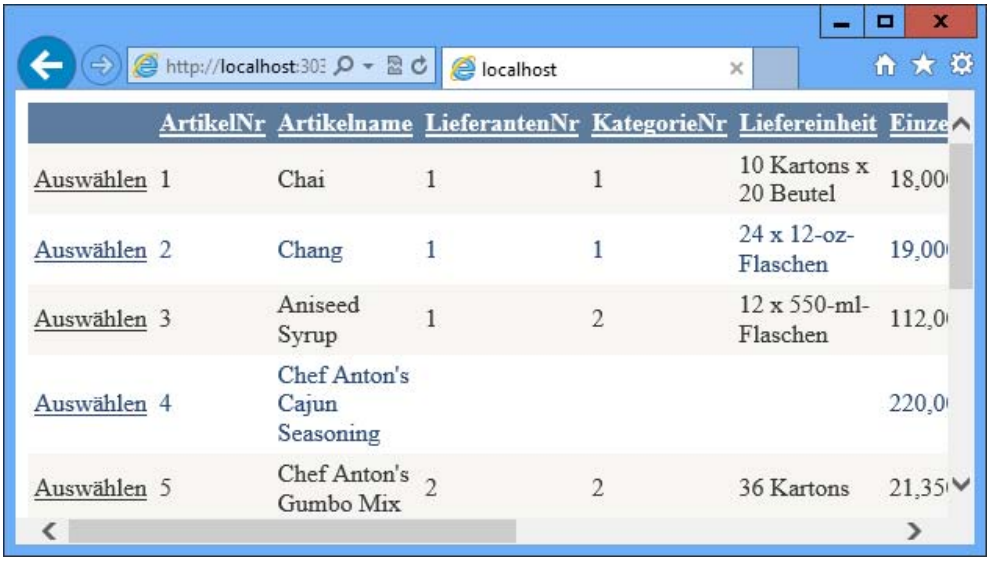
Nach dem Schließen des Assistenten sollten im *GridView* bereits die Tabellenspalten entsprechend der obigen Auswahl angezeigt werden.

Öffnen Sie noch kurz den *GridView*-Aufgabenbereich und aktivieren Sie die Optionen *Paging*, *Sortieren* und *Auswahl*:



Test

Wenn Sie wie gewohnt starten wollen (F5), werden Sie in einem Meldungsfenster zunächst aufgefordert, das Debuggen zu aktivieren. Nach dem *OK* können Sie im Internet-Explorer bereits die erste Tabellenansicht bewundern:



	ArtikelNr	Artikelname	LieferantenNr	KategorieNr	Liefereinheit	Einze
Auswählen	1	Chai	1	1	10 Kartons x 20 Beutel	18,00
Auswählen	2	Chang	1	1	24 x 12-oz-Flaschen	19,00
Auswählen	3	Aniseed Syrup	1	2	12 x 550-ml-Flaschen	112,0
Auswählen	4	Chef Anton's Cajun Seasoning				220,0
Auswählen	5	Chef Anton's Gumbo Mix	2	2	36 Kartons	21,35

Auch die Navigation zwischen den einzelnen Seiten ist bereits voll funktionstüchtig, klicken Sie ruhig mal auf eine der Ziffern im unteren Bereich des *GridView*-Steuerelements. Gleiches trifft auf die Möglichkeit zu, die Daten nach bestimmten Spalten zu sortieren, ein Klick auf den gewünschten Tabellenkopf-Eintrag genügt.

Ohne eine einzige Zeile Code geschrieben zu haben, liegt bereits eine voll funktionsfähige Anwendung vor!

Für Freunde des Drag&Drop eine gute Nachricht:

HINWEIS: Haben Sie im Server-Explorer bereits eine Verbindung definiert, können Sie die darin enthaltenen Tabellen/Views einfach per Drag&Drop in das WebForm ziehen. Alle erforderlichen Steuerelemente und Optionen werden dann automatisch erzeugt.

Nach diesem kleinen Ausflug in die Praxis wenden wir uns wieder der Theorie zu.

41.2 Einführung

Der kleine Ausflug in die Praxis hat Sie sicher von den Vorzügen von ASP.NET überzeugt. Während Sie früher noch reichlich Hand anlegen mussten, um die Daten auch in die Steuerelemente zu laden (von Sortieren, Paging etc. ganz zu schweigen), ersparen Ihnen die ASP.NET-Datensteuerelemente derartige Routineaufgaben (fast) völlig.

Übernommen werden die folgenden Aufgaben:

- Auswahl und Anzeige der Daten,
- Editieren, Hinzufügen und Löschen von Datensätzen,
- Sortieren, Filtern und Paging,
- Unterstützung bei Master-Detail-Darstellungen,
- Caching.

41.2.1 Konzept

Außer mit den bereits genannten *DataSource*-Steuerelementen haben Sie es in ASP.NET 4.5 im Wesentlichen noch mit den folgenden visuellen Steuerelementen zu tun:

- *GridView* (tabellarische Ansicht),
- *DetailsView* (einfache formularbasierte Ansicht eines Datensatzes),
- *FormView* (formularbasierte Ansicht mit Templates),
- *DataList* (tabellarische Ansicht mit Templates).
- *Chart* (Anzeige von Diagrammen)

Auch

- *TreeView*,
- *ListBox,*

- *DropDownList*
- sowie *CheckBoxList* und *RadioButtonList*

lassen sich für die Datenanzeige nutzen.

HINWEIS: Das gute alte *DataGrid* ist auch noch nicht ausgestorben, es fristet ein Gnadendasein auf Ihrer Festplatte, nicht jedoch im Werkzeugkasten.

Die grundsätzliche Vorgehensweise reduziert sich auf das Erstellen einer *DataSource* (mit enthaltenen SELECT, INSERT, UPDATE und DELETE-Befehlen) und deren Kopplung mit den visuellen Komponenten über die *DataSourceId*.

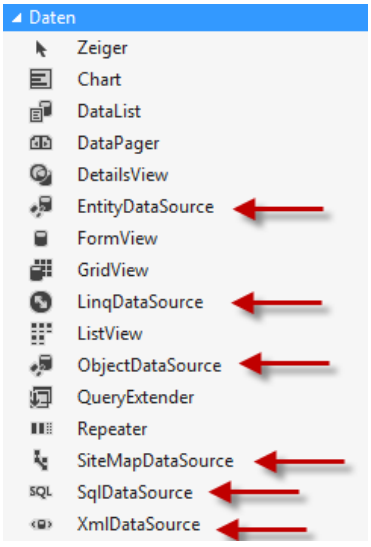
Mit *DataSet*s, *Connection*-Objekten, *DataReader*n etc. kommen Sie überhaupt nicht mehr in Kontakt[1]. Wer will, kann nach wie vor auch die alten Technologien einsetzen, wie das Einführungsbeispiel gezeigt hat.

Im Folgenden werden wir zunächst die einzelnen Steuerelemente unter die Lupe nehmen, bevor wir uns einigen speziellen Techniken (1 n-Beziehungen, Filtern etc.) zuwenden.

41.2.2 Übersicht über die DataSource-Steuerelemente

Bisher war immer nur von den *DataSource*-Steuerelementen die Rede, ohne im Einzelnen auf die verschiedenen Typen einzugehen.

Nachdem mit Visual Studio 2005 bereits eine Fülle von DataSource-Steuerelementen eingeführt wurde, beschränkte sich Visual Studio 2010 auf die *EntityDataSource*.

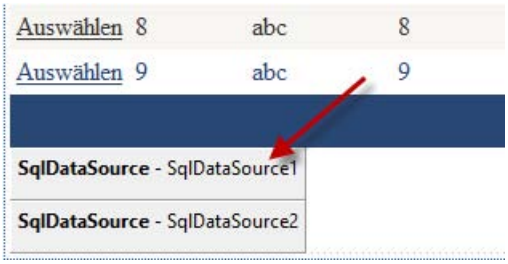


[1] Es gibt auch die Eigenschaft *DataSourceMode*, mit der Sie zwischen der Verwendung eines *DataSet*s und eines *DataReader*s umschalten können.

ASP.NET 4.5 bzw. Visual Studio 2012 geht jetzt wieder in die andere Richtung:

HINWEIS: Die *AccessDataSource* wurde aus dem Werkzeugkasten verbannt, ist aber zur Not noch auf Ihrer Festplatte zu finden. Der Grund ist laut Microsoft die mangelnde Sicherheit und Performance von Access-Datenbanken.

Alle obigen Steuerelemente sind zur Laufzeit nicht sichtbar, werden jedoch in der Entwicklungsumgebung direkt im Entwurfsbereich der WebForm angezeigt:



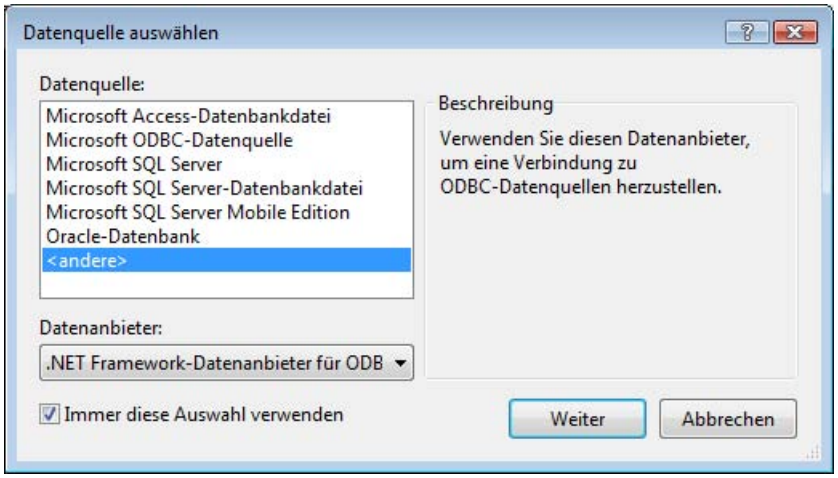
Hier wäre eine optische Trennung zwischen visuellen und nicht visuellen Komponenten (wie bei den Windows Forms) sicher sinnvoll.

Folgende Datenquellen sind derzeit verfügbar:

Typ	Bemerkung
SqlDataSource	Universeller Zugriff auf ADO.NET-Provider (z.B. MS SQL Server, SQLExpress, OLEDB, ODBC etc.)
ObjectDataSource	Zugriff auf Datenzugriffs-Ebene oder Business Objects
LinqDataSource	Zugriff auf Datenmengen, die das Ergebnis von LINQ-to SQL-Abfragen sind. Der Vorteil ist die Abstraktion und die einheitliche Abfragesprache für unterschiedliche Datenmengen.
EntityDataSource	Anbindung an eine Entity Data Model (EDM).
AccessDataSource	Nur noch für die ASP.NET 4.0-Kompatibilität: Zugriff auf Microsoft Access-Datenbanken
SiteMapDataSource	Zugriff auf *Web.SiteMap* zum Abrufen der Web-Strukturinformationen
XmlDataSource	Zugriff auf strukturierte XML-Daten

41.3 SQLDataSource

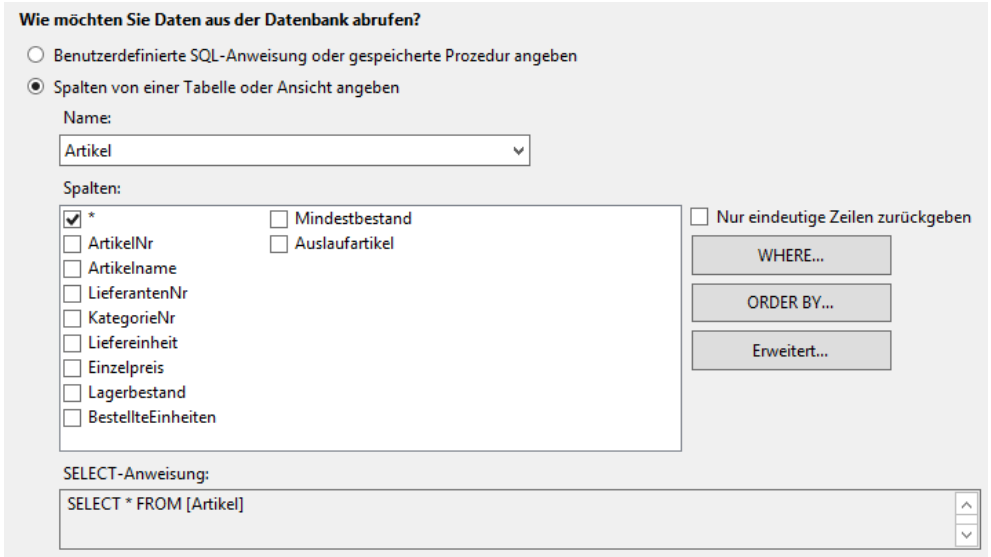
Wie in der obigen Tabelle bereits angedeutet, beschränkt sich der Zugriff dieses Steuerelements nicht nur auf den SQL Server, wie es auch die folgende Abbildung zeigt:



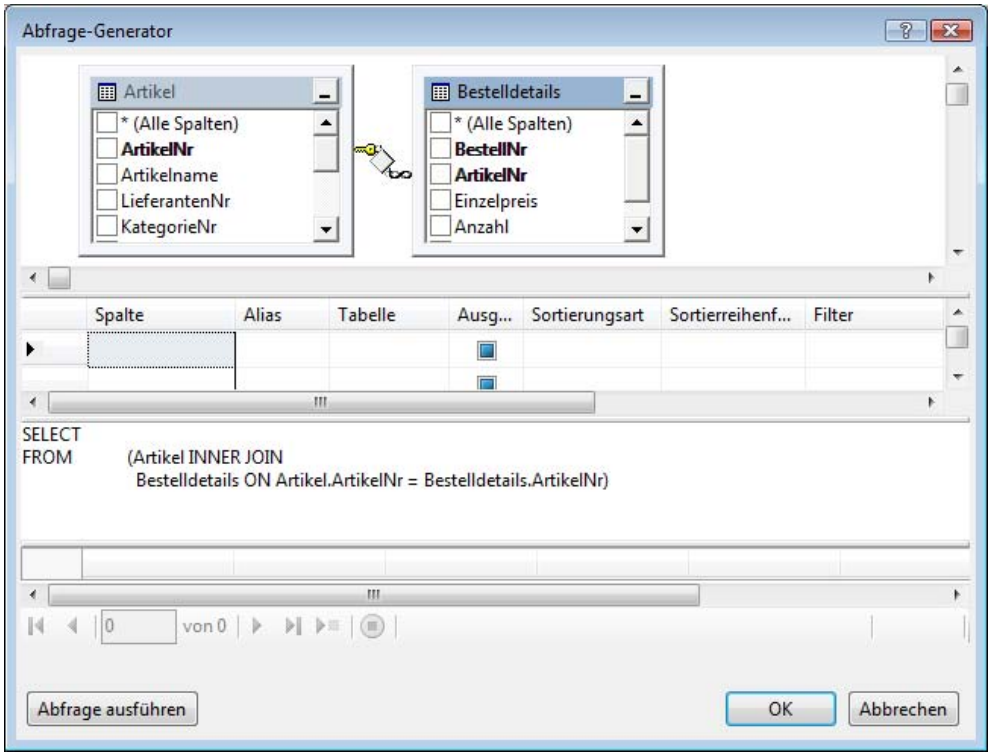
Fast alle Parameter der *SQLDataSource* lassen sich bequem über das Aufgaben-Menü bzw. den entsprechenden Assistenten setzen.

Nach der Auswahl der eigentlichen Datenquelle (siehe obige Abbildung) müssen Sie sich zunächst um die Datenauswahl (SELECT) kümmern. Dabei haben Sie die Wahl zwischen

- Tabellen/Views und
- selbstdefinierten SQL-Abfragen/Prozeduren.



Ein leistungsfähiger Query Builder (Abfrage-Generator) hilft Ihnen optional über die Hürden der SQL-Programmierung:



HINWEIS: Soll die Datenquelle editierbar sein (Einfügen, Ändern, Löschen), müssen Sie INSERT-, UPDATE- und DELETE-Abfragen definieren.

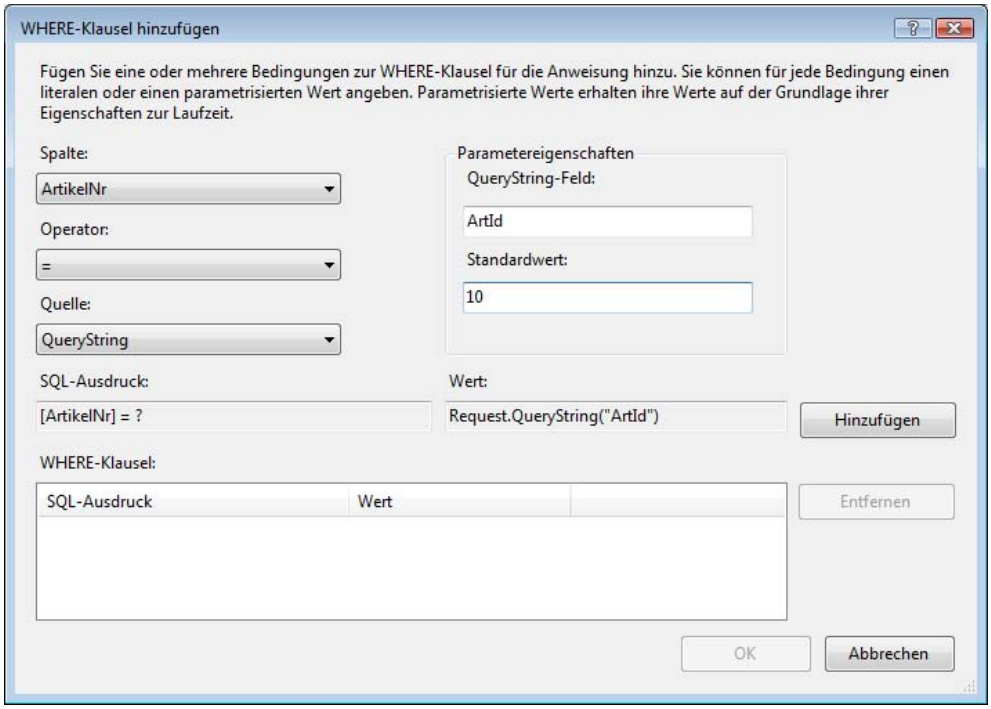
41.3.1 Datenauswahl mit Parametern

Nicht immer und in jedem Fall möchten Sie alle Daten aus einer Tabelle abrufen. Dass Sie mit Hilfe der WHERE-Klausel und der Verwendung von Parametern diese Aufgabe lösen können, dürfte dem SQL-Kundigen bekannt sein.

Neu ist jedoch die Möglichkeit, die Werte der Parameter direkt mit anderen Objekten bzw. Steuerelemente-Eigenschaften zu verknüpfen.

Als Quelle für die Parameter kommen folgende Möglichkeiten in Betracht:

- *Control* (Steuerelement-Id),
- *Cookie* (Name),
- *Form* (Formularfeld),
- *Profile* (Eigenschaftenname),
- *QueryString* (QueryString-Feld),
- *Session* (Sitzungsfeld).



Damit genügt schon ein erneuter Seitenaufruf mit angehängtem QueryString, um zum Beispiel Detaildaten anzuzeigen oder die Anzeige zu beschränken. Und das Beste bei alledem: Sie brauchen keine einzige Zeile Quellcode zu schreiben!

41.3.2 Parameter für INSERT, UPDATE und DELETE

Ähnlich wie beim SELECT funktioniert es auch mit dem Löschen, Einfügen und Ändern. In allen Fällen bietet die jeweilige Eigenschaft auch die Möglichkeit, Parameter über die o.g. Quellen zu setzen.

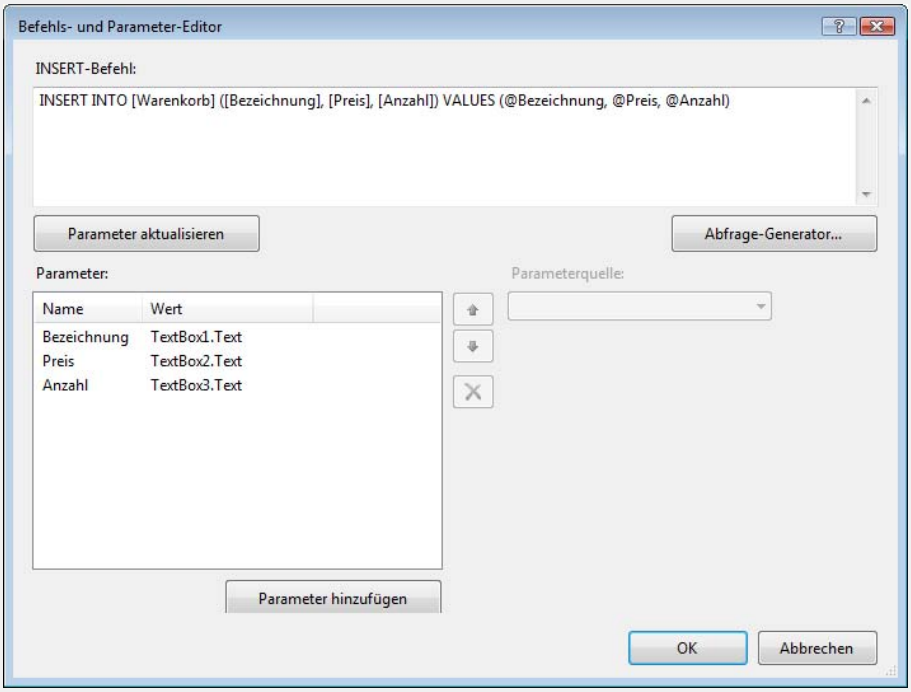
BEISPIEL 41.1: Umsetzung INSERT

C#

In die Tabelle *Warenkorb* soll per Button-Klick ein neuer Datensatz eingefügt werden. Die Werte für die Spalten *Bezeichnung*, *Preis* und *Anzahl* sollen direkt aus *TextBox1 ... TextBox3* ausgelesen werden.

Die Einstellungen für die *SqlDataSource.InsertQuery* entnehmen Sie bitte der folgenden Abbildung:

BEISPIEL 41.1: Umsetzung INSERT



Das eigentliche Einfügen zur Laufzeit ist jetzt kinderleicht:

```
protected void Button1_Click(object sender, EventArgs e)
 {
     SqlDataSource2.Insert();
 }
```

Alternativ können Sie natürlich auch eine Stored Procedure verwenden, in diesem Fall tragen Sie nur den Namen der Prozedur ein und klicken nachfolgend auf den Button *Parameter aktualisieren*.

BEISPIEL 41.2: Verwendung einer Stored Procedure

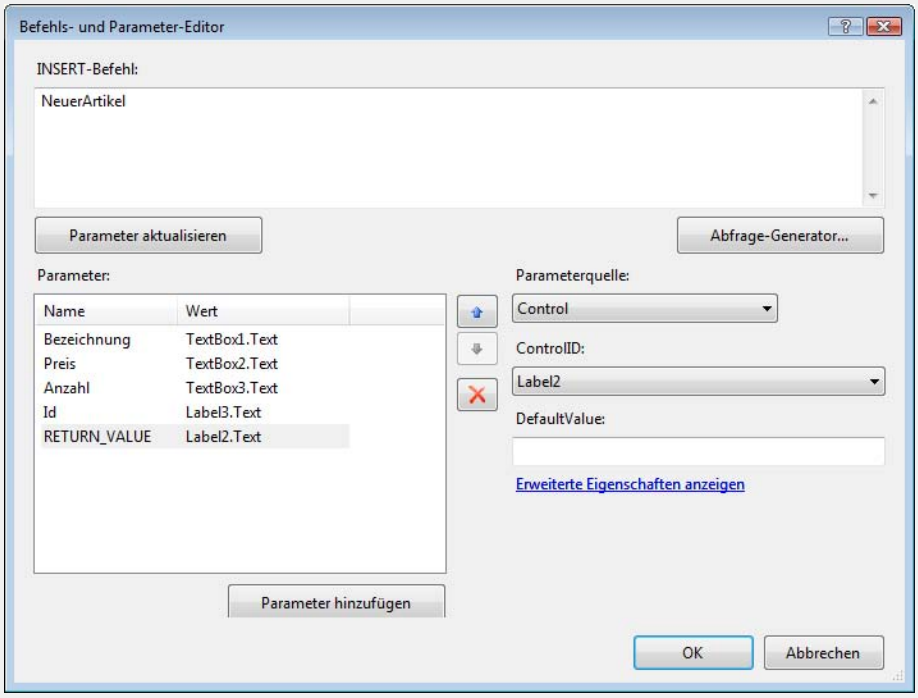
Die Stored Procedure:

```
CREATE PROCEDURE dbo.NeuerArtikel (
 @Bezeichnung varchar(50),
 @Preis float,
 @Anzahl int,
 @Id int OUTPUT
 )
AS
 INSERT INTO [Warenkorb]
 ([Bezeichnung], [Preis], [Anzahl]) VALUES (@Bezeichnung, @Preis, @Anzahl);
```

BEISPIEL 41.2: Verwendung einer Stored Procedure

```
Set @Id = @@Identity
RETURN 0
```

Die abgerufenen Parameter:



41.3.3 Methoden

Ein Blick in die Liste der *SqlDataSource*-Methoden dürfte zunächst für Ernüchterung sorgen, außer dem schon erwähnten *Insert* finden sich lediglich

- *Select*,
- *Delete*,
- *Update*
- und *DataBind*.

Wer jetzt enttäuscht ist, sei getröstet. Mit diesen wenigen Methoden können Sie alle relevanten Aufgaben lösen!

HINWEIS: Wer immer noch verzweifelt nach Navigations-Methoden (*First*, *Next* etc.) Ausschau hält, sei daran erinnert, dass wir hier mit Datenmengen arbeiten, ein Datensatzzeiger im althergebrachten Sinn existiert nicht mehr.

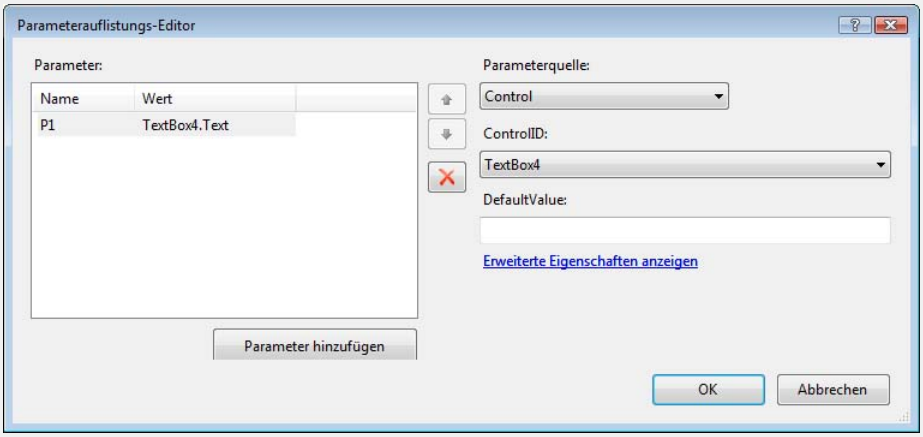
Auch wenn die Verwendung einer Eigenschaft etwas unglücklich ist, mit dem Setzen der Eigenschaft *FilterExpression* können Sie die bereits vorselektierte (SELECT ... WHERE) Datenmenge filtern. Auch hier lassen sich Parameter verwenden, die wiederum an Quellen wie Controls etc. gebunden werden können. Allerdings werden hier die Parameter durchnummeriert und nicht benannt.

BEISPIEL 41.3: Filtern der Datenmenge

```
protected void Button2_Click(object sender, EventArgs e)
{
    SqlDataSource2.FilterExpression = "Bezeichnung LIKE '{0}'";
}
```

Wie Sie sehen, brauchen Sie für die Filterausdrücke nicht gleich eine neue Programmiersprache zu lernen, einige SQL-Kenntnisse (WHERE) sind hingegen ganz hilfreich.

Die zugehörige *FilterParameters*-Eigenschaft setzen Sie mittels Assistent:



41.3.4 Caching

Sinn macht die Verwendung von *FilterExpression* erst bei Verwendung der Caching-Funktionalität der *SqlDataSource*. Da die Daten komplett im Speicher liegen, muss bei geändertem Filterausdruck kein erneuter Datenbankzugriff vorgenommen werden.

Das Caching aktivieren Sie über die beiden folgenden Eigenschaften:

- *EnableCaching* (*True*/*False*)
- und *CacheDuration* (die Zeitdauer in Sekunden).

Nachfolgende Filterabfragen werden auf diese Weise beschleunigt, der Server selbst wird entlastet (genügend Speicher vorausgesetzt).

41.3.5 Aktualisieren/Refresh

Bei Ihnen ist vielleicht schon die Frage aufgetaucht, wie auf Datenänderungen per externem SQL-Befehl (z.B. andere Connection) reagiert werden kann. Ein direkter Refresh-Befehl ist zwar nicht vorhanden, mit dem Aufruf der Methode *DataBind* haben Sie jedoch auch dieses Problem im Griff.

41.4 AccessDataSource

Wie der Name schon vermuten lässt, kommt dieses Steuerelement beim Zugriff auf Access-Datenbanken zum Einsatz. Nun kann man über die Bedeutung von Access-Datenbanken in Web-Applikationen sicher geteilter Meinung sein, für einige Aufgaben genügte Access jedoch in jedem Fall.

HINWEIS: Mit der Einführung des SQL Server 2012 Express LocalDB bietet sich jetzt jedoch eine starke Alternative im Bereich der Desktop-Datenbanken. Sie sollten im Zweifel also darüber nachdenken, Ihre Access-Datenbank in eine LocalDB-Datenbank zu portieren und auf Access und dieses Control zu verzichten.

Wer noch ein Altprojekt pflegt oder wer partout noch bei Access bleiben will, kann dieses Control als Alternative zur *SqlDataSource* nutzen, auch wenn beide Controls über fast den selben Funktionsumfang verfügen. Die wesentlichste Erweiterung ist die Eigenschaft *DataFile*, die den Pfad zur Access-Datenbank angibt, dafür ist der Zugriff auf den Connectionstring nicht mehr möglich.

Aus diesen Gründen können wir Sie zu allen Fragen – ohne schlechtes Gewissen – auf den vorhergehenden Abschnitt verweisen.

41.5 ObjectDataSource

Oh je, noch eine Datenquelle und noch dazu mit Objekten, werden sicher einige stöhnen. Doch gerade die Puristen und Verfechter der reinen Lehre sollten sich dieses Steuerelement näher ansehen, bietet es doch die Möglichkeit, mit "wenig Aufwand" Geschäftsobjekte an die neuen Oberflächen-Steuerelemente zu binden, d h., zwischen Oberfläche und Datenbank wird eine zusätzliche Schicht geschoben, die logische Aufgaben übernimmt und das Frontend von der eigentlichen Datenbasis abschirmt.

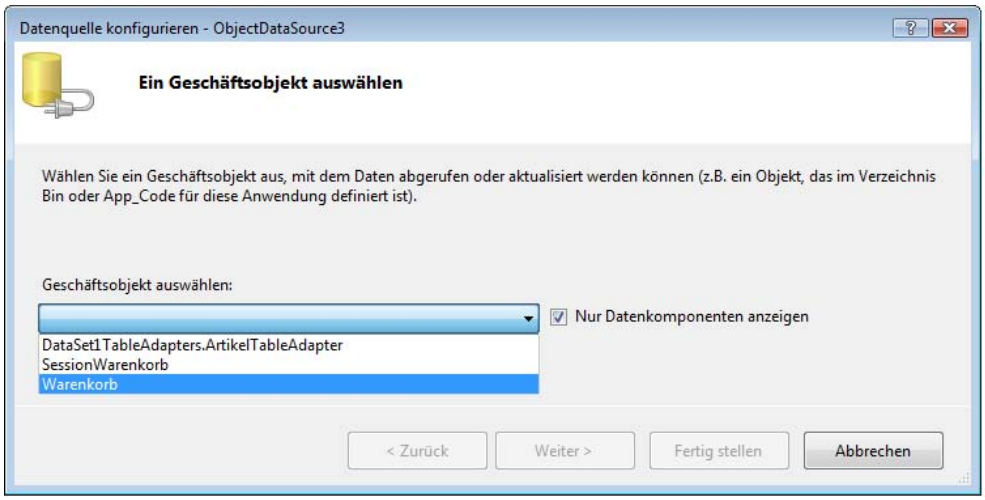
41.5.1 Verbindung zwischen Objekt und DataSource

Die Schnittstelle zwischen Geschäftslogik (Objekt) und der *ObjectDataSource* wird durch einige vordefinierte Methoden realisiert:

- SELECT (Rückgabewerte z.B. Objekte, Collections, DataSets),
- UPDATE (Übergabewert z.B. Objekte oder einzelne Parameter),
- INSERT (Übergabewert z.B. Objekte oder einzelne Parameter) und
- DELETE (Übergabewert z.B. Objekte oder einzelne Parameter).

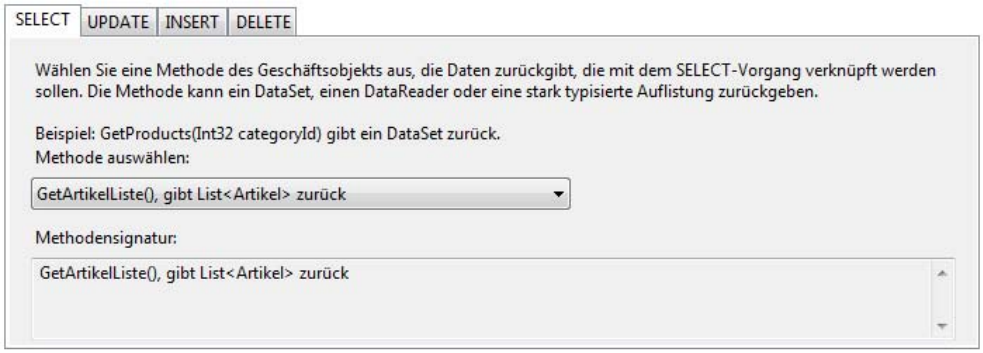
Die Zuordnung kann entweder per Assistent oder über die Eigenschaften *SelectMethod* ... *DeleteMethod* erfolgen.

Doch bevor es soweit ist, muss das eigentliche Geschäftsobjekt ausgewählt werden. Dazu ist es wichtig, dass dieses sich im Verzeichnis *App_Code* befindet, wenn Sie mit einem ASP.NET-Website-Projekt arbeiten[1].



Im obigen Beispiel wählen wir das Geschäftsobjekt *Warenkorb*, auf die Implementierung kommen wir im Verlauf des Abschnitts zurück.

Im nächsten Schritt können Sie dann die einzelnen Zugriffs-Methoden zuweisen:



[1] Sie erinnern sich: derartige Projekte werden in mehrere unabhängige Einzel-Assemblies kompiliert, das App-Code-Verzeichnis ist die gemeinsame Basis für übergreifende Klassen.

41.5.2 Ein Beispiel sorgt für Klarheit

Im Folgenden wollen wir über ein Geschäftsobjekt *Warenkorb* Daten in der Tabelle *Warenkorb* bearbeiten. Die Geschäftslogik beschränken wir auf den reinen Datentransfer zwischen Oberfläche und Datenbank, zusätzliche Funktionen wie Rechteverwaltung, Integritätsprüfungen etc. würden an dieser Stelle nur verwirren.

Die zugrunde liegende SQL-Server-Tabelle:

	Spaltenname	Datentyp	NULL zula...
🔑	Id	int	☐
	Bezeichnung	varchar(50)	☑
	Preis	float	☑
	Anzahl	int	☑
▶			☐

Das Geschäftsobjekt *Warenkorb* soll eine streng typisierte Liste von *Artikel*-Objekten bereitstellen. Die entsprechende Klasse müssen wir zunächst definieren:

```
public class Artikel
{
    private String _Bezeichnung;
    private Double _Preis;
    private int _Anzahl;
    private int _Id;

   public Artikel()
   { }

    public Artikel(String Bezeichnung, Double Preis, int Anzahl, int ID)
    {
        this._Bezeichnung = Bezeichnung;
        this._Preis = Preis;
        this._Anzahl = Anzahl;
        this._Id = ID;
    }

    public string Bezeichnung
    {
        get {return this._Bezeichnung;}
        set {this._Bezeichnung = value;}
    }

    public double Preis
    {
        get {return this._Preis;}
        set {this._Preis = value;}
    }
```

```
    public int Anzahl
    {
        get {return this._Anzahl;}
        set {this._Anzahl = value; }
    }

    public int Id
    {
        get {return this._Id; }
        set {this._Id = value; }
    }
}
```

HINWEIS: Vergessen Sie nicht, die Klasse im Verzeichnis *App_Code* zu speichern, anderenfalls haben Sie keinen Zugriff darauf.

Unser Geschäftsobjekt *Warenkorb*:

```
...
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
using System.ComponentModel;

[DataObject(true)]
public class Warenkorb
{
```

Die interne Liste:

```
    private  List<Artikel> _ArtikelListe;
```

Im Konstruktor wird zunächst die Liste erzeugt:

```
    public Warenkorb()
    {  _ArtikelListe = new List<Artikel>();   }
```

Die Methode *GetArtikelListe* gibt die Liste von *Artikel*-Objekten zurück (SELECT):

```
    [DataObjectMethod(DataObjectMethodType.Select)]
    public  List<Artikel> GetArtikelListe()
    {
```

Auslesen aus der Datenbank:

```
        SqlConnection conn = new SqlConnection(ConfigurationManager.ConnectionStrings["
                                DatenConnectionString2"].ConnectionString);
        SqlCommand cmd = new SqlCommand("SELECT * FROM warenkorb", conn);
        conn.Open();
        SqlDataReader dr = cmd.ExecuteReader();
```

Einlesen in die interne Liste:

```
        while (dr.Read())
            _ArtikelListe.Add(new Artikel((string) dr["Bezeichnung"],
                (double) dr["Preis"], (int) dr["Anzahl"],(int) dr["id"] ));
        dr.Close();
        conn.Close();
        return _ArtikelListe;
    }
```

HINWEIS: Das Objekt muss *stateless* programmiert werden, da es mit jedem Methodenaufruf neu erzeugt wird. D h., insbesondere die Persistenz muss bei DELETE, UPDATE und INSERT **sofort** hergestellt werden.

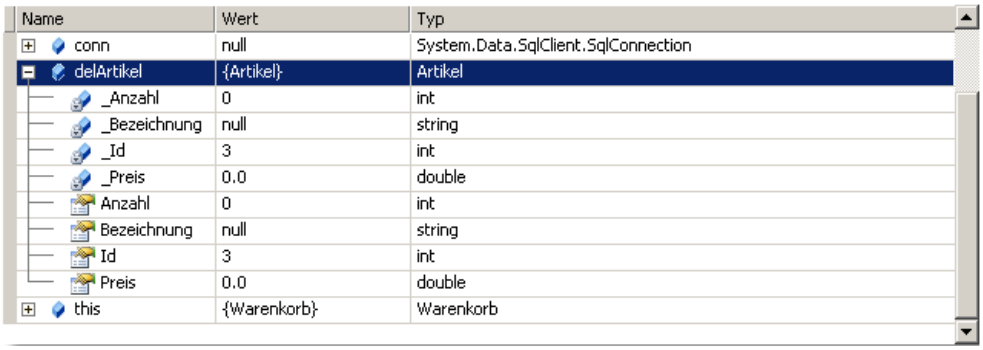
Der per Parameter übergebene *Artikel* soll aktualisiert werden:

```
    [DataObjectMethod(DataObjectMethodType.Update)]
    public void UpdateArtikel(Artikel updArtikel)
    {
        SqlConnection conn = new SqlConnection(ConfigurationManager.Connection-
                             Strings["DatenConnectionString2"].ConnectionString);
        SqlCommand cmd = new SqlCommand("UPDATE warenkorb SET [Bezeichnung]=@bez," +
                       "[Preis]=@preis, [Anzahl]=@anz WHERE [ID]=@id", conn);
        cmd.Parameters.AddWithValue("@bez", updArtikel.Bezeichnung);
        cmd.Parameters.AddWithValue("@preis", updArtikel.Preis);
        cmd.Parameters.AddWithValue("@anz", updArtikel.Anzahl);
        cmd.Parameters.AddWithValue("@id", updArtikel.Id);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }
```

Eine Artikel löschen:

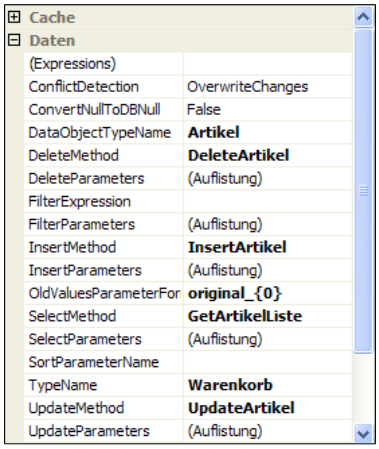
```
    [DataObjectMethod(DataObjectMethodType.Delete)]
    public  void DeleteArtikel(Artikel delArtikel)
    {
        SqlConnection conn = new SqlConnection(ConfigurationManager.Connection-
                             Strings["DatenConnectionString2"].ConnectionString);
        SqlCommand cmd = new SqlCommand("DELETE FROM warenkorb WHERE [ID]=@id", conn);
        cmd.Parameters.AddWithValue("@id", delArtikel.Id);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }
```

HINWEIS: Auch wenn hier ein komplettes Objekt übergeben wird, bei einem späteren Zugriff durch die *ObjectDataSource* ist nur die ID richtig gesetzt, wie die folgende Abbildung zeigt:



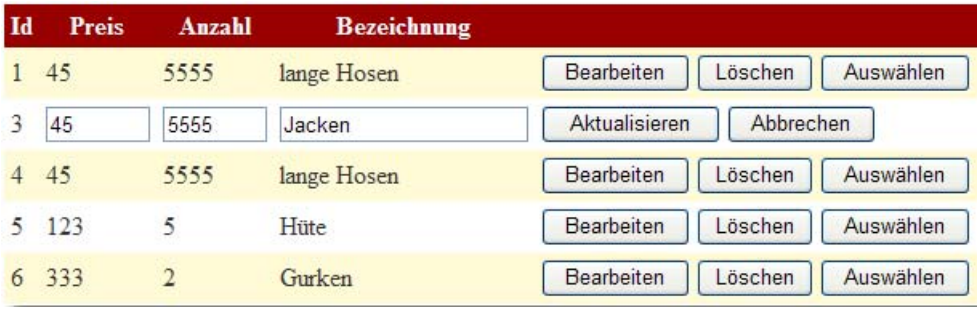
Name	Wert	Typ
conn	null	System.Data.SqlClient.SqlConnection
delArtikel	{Artikel}	Artikel
_Anzahl	0	int
_Bezeichnung	null	string
_Id	3	int
_Preis	0.0	double
Anzahl	0	int
Bezeichnung	null	string
Id	3	int
Preis	0.0	double
this	{Warenkorb}	Warenkorb

Nach Zuweisung des Geschäftsobjekts *Warenkorb* an die *ObjectDataSource* und der Zuordnung der einzelnen Methoden (siehe folgende Abbildung),



Cache	
Daten	
(Expressions)	
ConflictDetection	OverwriteChanges
ConvertNullToDBNull	False
DataObjectTypeName	**Artikel**
DeleteMethod	**DeleteArtikel**
DeleteParameters	(Auflistung)
FilterExpression	
FilterParameters	(Auflistung)
InsertMethod	**InsertArtikel**
InsertParameters	(Auflistung)
OldValuesParameterFor	**original_{0}**
SelectMethod	**GetArtikelListe**
SelectParameters	(Auflistung)
SortParameterName	
TypeName	**Warenkorb**
UpdateMethod	**UpdateArtikel**
UpdateParameters	(Auflistung)

können Sie sich mit einem *GridView* von der Funktionsfähigkeit unseres Geschäftsobjekts überzeugen:



Id	Preis	Anzahl	Bezeichnung			
1	45	5555	lange Hosen	Bearbeiten	Löschen	Auswählen
3	45	5555	Jacken	Aktualisieren	Abbrechen	
4	45	5555	lange Hosen	Bearbeiten	Löschen	Auswählen
5	123	5	Hüte	Bearbeiten	Löschen	Auswählen
6	333	2	Gurken	Bearbeiten	Löschen	Auswählen

Wie Sie sehen, lässt sich die Datenbank mit Hilfe der *ObjectDataSource* komplett kapseln, obige Datenpersistenz hätten Sie zum Beispiel auch mit einer XML-Datei oder auch einer *Session*-Variablen erreicht. Wer es nicht glaubt, sollte sich die folgende Implementierung näher ansehen.

41.5.3 Geschäftsobjekte in einer Session verwalten

Das *Artikel*-Objekt sowie die Schnittstelle von *Warenkorb* aus dem vorhergehenden Beispiel wird beibehalten, statt der SQL Server Datenbank verwenden wir eine *Session*-Variable. Der Nachteil (oder auch Vorteil): Nach einigen Minuten Inaktivität ist alles weg.

Aus Sicht des Programmierers müssen wir rudimentäre Datenbankfunktionen implementieren, was aber kaum ein Problem sein wird:

```
...
[DataObject(true)]
public class SessionWarenkorb
{
    private  List<Artikel> _ArtikelListe;
```

Der vereinfachte Zugriff auf das *Session*-Objekt:

```
    private System.Web.SessionState.HttpSessionState Session = HttpContext.Current.Session;
```

Im Konstruktor lesen wir entweder die Session aus,

```
    public SessionWarenkorb()
    {
        if (Session["Warenkorb"] != null)
        {
            _ArtikelListe = (List<Artikel>) Session["Warenkorb"];
        }
```

oder wir erzeugen eine neue Liste:

```
        else
        {
            _ArtikelListe = new List<Artikel>();
        }
    }
```

Die Auswahl ist jetzt besonders einfach:

```
    [DataObjectMethod(DataObjectMethodType.Select)]
    public  List<Artikel> GetArtikelListe()
    {
        return _ArtikelListe;
    }
```

Ein Update erfordert schon etwas Handarbeit, wir müssen den Eintrag suchen:

```
    [DataObjectMethod(DataObjectMethodType.Update)]
    public void UpdateArtikel(Artikel updArtikel)
    {
```

```
        foreach (Artikel art in _ArtikelListe)
        {
            if (art.Id == updArtikel.Id)
            {
                art.Bezeichnung = updArtikel.Bezeichnung;
                art.Preis = updArtikel.Preis;
                art.Anzahl = updArtikel.Anzahl;
                break;
            }
        }
        Session["Warenkorb"] = _ArtikelListe;
    }
```

Auch beim Delete sind wir auf Kenntnisse der Collection-Programmierung angewiesen:

```
    [DataObjectMethod(DataObjectMethodType.Delete)]
    public  void DeleteArtikel(Artikel delArtikel)
    {
        for (int i = 0; i < _ArtikelListe.Count; i++)
             if (_ArtikelListe[i].Id == delArtikel.Id)
             {
                 _ArtikelListe.RemoveAt(i);
                 break;
             }
         Session["Warenkorb"] = _ArtikelListe;
     }
```

Im Unterschied zum Geschäftsobjekt aus dem Vorgängerbeispiel erzeugen wir auch neue Datensätze:

```
    [DataObjectMethod(DataObjectMethodType.Insert)]
    public  void InsertArtikel(Artikel insArtikel)
    {
        int Counter = 0;
        foreach (Artikel art in _ArtikelListe)
        {
            if (art.Id > Counter) Counter = art.Id;
        }
        Counter++;
        _ArtikelListe.Add(new Artikel(insArtikel.Bezeichnung, insArtikel.Preis,
                          insArtikel.Anzahl, Counter));
        Session["Warenkorb"] = _ArtikelListe;
    }
}
```

Binden Sie dieses Objekt, wie schon beim letzten Beispiel, an eine *ObjectDataSource* und fügen Sie außer einem *GridView* auch ein *DetailsView* ein (für das Erzeugen neuer Datensätze setzen Sie *DefaultMode=Insert*).

HINWEIS: Damit das Löschen auch funktioniert, müssen Sie beim *GridView* die Eigenschaft *DataKeyNames* auf *Id* setzen.

Ein Test zeigt, dass es funktioniert:

41.6 SitemapDataSource

Diese Datenquelle beschränkt sich auf die Anzeige einer hierarchischen Sitemap-Datei (Default *Web.SiteMap*). Über die verfügbaren Eigenschaften können Sie Einfluss auf die Abbildung dieser Datei in den gebundenen Controls nehmen.

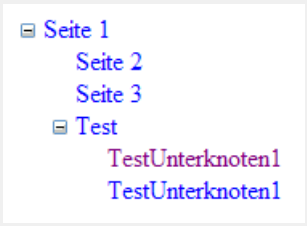
BEISPIEL 41.4: ***Web.SiteMap***

XML

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
    <siteMapNode url="Seite1.asp" title="Seite 1"  description="">
        <siteMapNode url="Seite2.aspx" title="Seite 2"  description="" />
        <siteMapNode url="Seite3.aspx" title="Seite 3"  description="" />
        <siteMapNode url="Test.aspx" title="Test"  description="">
            <siteMapNode url="TestUnterknoten1.aspx" title="TestUnterknoten1"/>
            <siteMapNode url="TestUnterknoten2.aspx" title="TestUnterknoten1"/>
        </siteMapNode>
    </siteMapNode>
</siteMap>
```

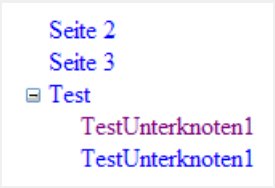
Ergebnis

Eine gebundene *TreeView* wird folgende Daten anzeigen:



Ändern Sie die Eigenschaft *ShowStartingNode* auf *False*, wird der Root-Knoten nicht mehr angezeigt:

BEISPIEL 41.4: ***Web.SiteMap***



HINWEIS: Mehr zum Thema *SiteMap* finden Sie im Kapitel 40.

41.7 LinqDataSource

Neben der "trivialen" Variante, eine einfache Collection mittels LINQ zu verarbeiten und an Controls zu binden, bietet sich vor allem die Verwendung im Zusammenhang mit LINQ to SQL an. In diesem Fall ist es auch problemlos möglich, neue oder geänderte Datensätze an die Datenquelle zu senden bzw. Datensätze zu löschen. Hier spielt LINQ seine Vorteile klar aus: Sie brauchen sich nicht um die Hintergründe zu kümmern, es genügt, wenn Sie die entsprechende Funktion über die Eigenschaften *EnableDelete, EnableInsert, EnableUpdate* der *LinqDataSource* freischalten.

41.7.1 Bindung von einfachen Collections

Ausgehend von einer trivialen Klasse *Daten,* die über die Eigenschaft *Mitarbeiter* eine Liste von Strings bereitstellt, wollen wir die Verwendung der *LinqDataSource* demonstrieren.

Zunächst die Klassendefinition (Achten Sie darauf, dass in einem Website-Projekt die Klasse im Ordner *App_Code* gesichert wird.):

```
using System;
...
public class Daten
{
    string[] _Mitarbeiter = {"Müller", "Meyer", "Lehmann", "Schulze" };

    public string[] Mitarbeiter
    {
        get
        {
            return _Mitarbeiter;
        }
    }
    public Daten()
    { }
}
```

Die Konfiguration der *LinqDataSource* gestaltet sich recht simpel (ASP.NET-Quellcode):

```
    <asp:LinqDataSource ID="LinqDataSource1" runat="server"
        ContextTypeName="Daten"
        TableName="Mitarbeiter">
    </asp:LinqDataSource>
```

Über die Eigenschaft *ContextTypeName* weisen Sie zunächst die gewünschte Klasse zu, die eigentliche Datenmenge (Collection) bestimmen Sie mit der *TableName*-Eigenschaft.

HINWEIS: Natürlich könnten Sie dieses Zuweisung, wie bei der *ObjectDataSource,* auch per Assistent realisieren, aber das dürfte sich bei lediglich zwei Eigenschaften kaum lohnen.

Die Anbindung, z.B. eines *GridView*-Controls, gestaltet sich im Weiteren recht einfach, weisen Sie die *DataSourceId* zu und setzen Sie die Eigenschaft *AutoGenerateColumns* auf *True* (Default-Wert)*:*

```
    <asp:GridView ID="GridView1" runat="server" DataSourceID="LinqDataSource1"
       CellPadding="4" ForeColor="#333333" GridLines="None">
...
    </asp:GridView>
```

Die Ansicht zur Laufzeit:

Item
Müller
Meyer
Lehmann
Schulze

41.7.2 Bindung eines LINQ to SQL-DataContext

HINWEIS: Wie Sie die Mapper-Klassen für eine LINQ to SQL-Abfrage erstellen, ist nicht Gegenstand dieses Kapitels.

Ausgehend von den per Assistent/Designer erstellten LINQ to SQL-Mapperklassen können Sie der *LinqDataSource* zunächst den übergeordneten *DataContext* per *ContextTypeName*-Eigenschaft zuweisen.

Danach bestimmen Sie per *TableName*-Eigenschaft, welche Tabelle abgefragt werden soll (entspricht der FROM-Klausel in LINQ). Die Auswahl einzelner Spalten realisieren Sie mit der *Select*-Eigenschaft, der Sie ein gültiges LINQ-Statement zuweisen.

BEISPIEL 41.5: Auswahl der Spalten *BestellNr* und *Einzelpreis*

HTML
```
    <asp:LinqDataSource ID="LinqDataSource2" runat="server"
        ContextTypeName="DatenDataContext"
        TableName="Bestelldetails">
        Select="new (BestellNr As BestNr, Einzelpreis AS EPreis)"
    </asp:LinqDataSource>
```

HINWEIS: Solange Sie sich auf vorhandene (einfache) Spalten beschränken (siehe oben), können Sie bei der Bindung von Controls auch die entsprechenden Assistenten zur Spaltenkonfiguration verwenden. Nehmen Sie aber berechnete Spalten oder Detaildaten ist Schluss mit lustig, und es bleibt nur der steinige Web über den ASP.NET-Quellcode.

BEISPIEL 41.6: Berechnete Spalte erzeugen

HTML
```
    <asp:LinqDataSource ID="LinqDataSource2" runat="server"
        ContextTypeName="DatenDataContext"
        TableName="Bestelldetails">
        Select="new (BestellNr As BestNr, Einzelpreis + Anzahl As GPreis)"
    </asp:LinqDataSource>
```

BEISPIEL 41.7: Zugriff auf Detaildaten

HTML
```
    <asp:LinqDataSource ID="LinqDataSource2" runat="server"
        ContextTypeName="DatenDataContext"
        TableName="Bestelldetails">
        Select="new (BestellNr As BestNr, Artikel.Artikelname AS AName)"
    </asp:LinqDataSource>
```

Hier sehen Sie auch gleich einen Vorteil der LINQ to SQL-Mapperklassen: Detaildaten müssen Sie nicht mühevoll per JOIN zuordnen, sondern Sie können dazu einfach die hierarchisch geschachtelten Eigenschaften verwenden (*Artikelname* ist eine Eigenschaft der Klasse *Artikel*, die dem Feld *Artikel* zugeordnet ist).

BEISPIEL 41.8: Anzeige der beiden obigen Felder in einem *GridView*

HTML
```
<asp:GridView ID="GridView3" runat="server" DataSourceID="LinqDataSource2"
     AutoGenerateColumns="False" CellPadding="4" ForeColor="#333333" GridLines="None">
     <FooterStyle BackColor="#1C5E55" Font-Bold="True" ForeColor="White" />
     <Columns>
           <asp:BoundField  DataField="AName" HeaderText="ArtikelName"  />
           <asp:BoundField DataField="GPreis" HeaderText="GesamtPreis"  />
       </Columns>
...
    </asp:GridView>
```

BEISPIEL 41.8: Anzeige der beiden obigen Felder in einem *GridView*

Ergebnis

Die zur Laufzeit angezeigte Tabelle:

ArtikelName	GesamtPreis
Queso Cabrales	26,0000
Singaporean Hokkien Fried Mee	19,8000
Mozzarella di Giovanni	29,3000
Tofu	27,6000

HINWEIS: Auf weitere Details zur *LinqDataSource* gehen wir in unserem Buch [Visual C# 2012 Datenbankprogrammierung] ein, da dort auch die notwendigen LINQ to SQL-Grundlagen erklärt werden.

41.8 EntityDataSource

An dieser Stelle haben wir es mit einem recht speziellen Vertreter der Gattung "Datenquelle" zu tun, ist dieses Control doch speziell für den Zugriff auf Entity Data Modelle angepasst.

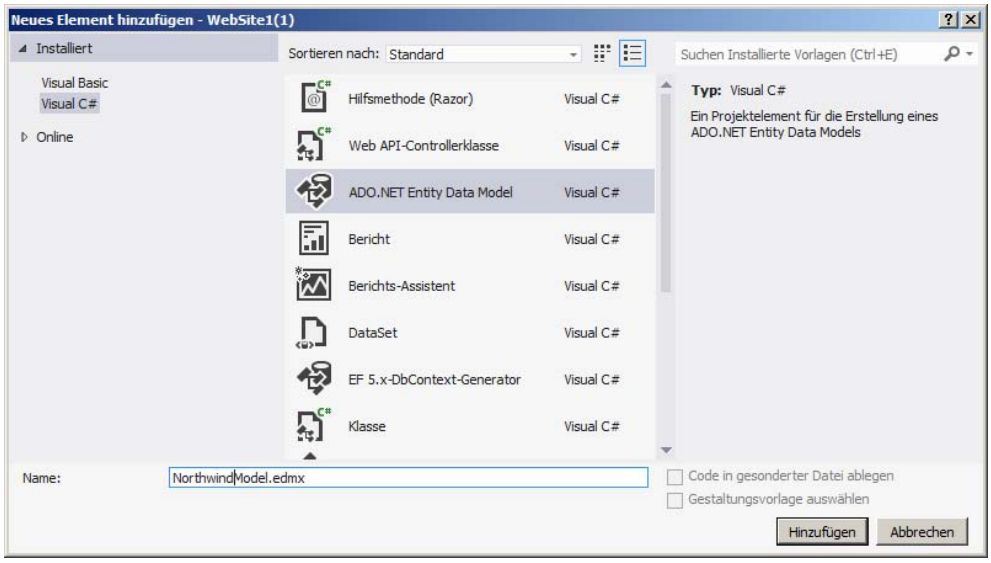
Die *EntityDataSource* wurde, etwas unbeachtet, bereits mit dem Service Pack 1 für das .NET-Framework 3.5 ausgeliefert, die Produktpolitik von Microsoft war in diesem Zusammenhang eher suboptimal.

HINWEIS: Da wir im Rahmen dieses Buchs nicht weiter auf die Arbeit mit dem EDM (*Entity Data Modell*) eingehen können, belassen wir es an dieser Stelle bei einem Beispiel. Auf weitere Details zur *EntityDataSource* gehen wir in unserem Buch [Visual C# 2012 Datenbankprogrammierung] ein, da dort auch die nötigen Grundlagen zum Entity Relationship Model (ERM) dargestellt werden.

Ausgangspunkt für die Arbeit mit dem Control ist zunächst eine geeignete Datenquelle, d h., ein vom Entity Framework unterstützter Datenprovider. Im folgenden Beispiel werden wir auf die "gute alte" *Northwind*-Datenbank des Microsoft SQL Servers zugreifen und versuchen, die Tabelle *Products* an ein *GridView* anzubinden.

41.8.1 Entity Data Model erstellen

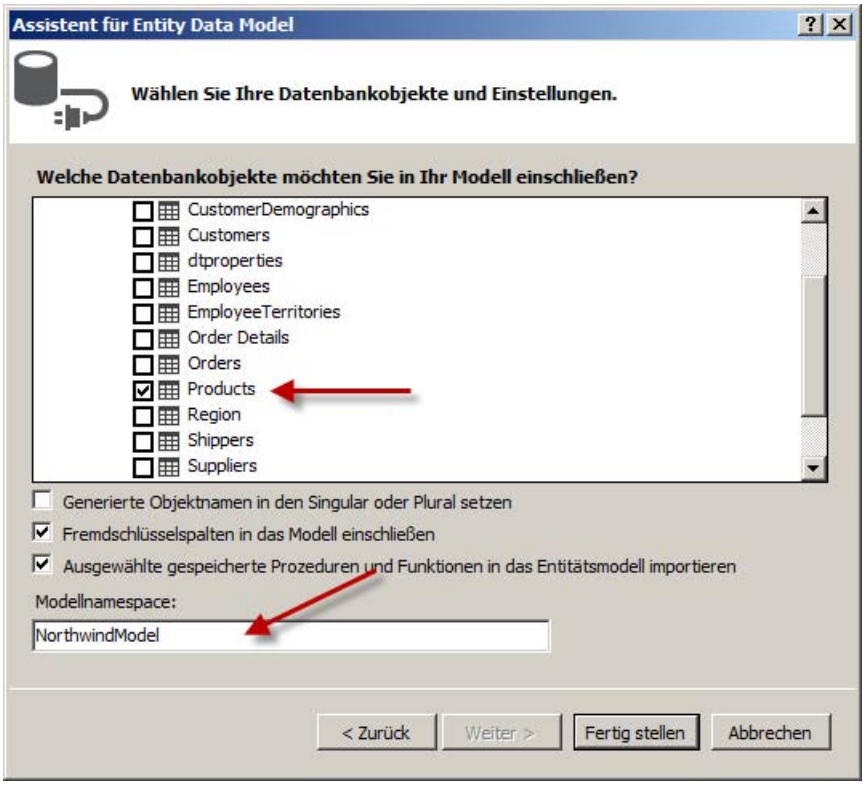
Bevor wir uns um Oberfläche etc. kümmern, brauchen wir ein funktionierendes Datenmodell. Erstellen Sie dieses über *Hinzufügen|Neues Element|ADO.NET Entity Data Model*:



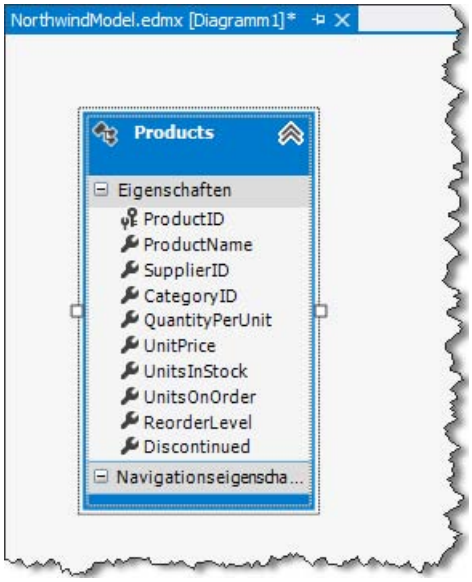
Vergeben Sie gleich hier den Namen *NorthwindModel.edmx* und klicken Sie dann auf "Hinzufügen". Im weiteren Verlauf müssen Sie sich entscheiden, ob Sie ein Modell aus einer bestehenden Datenbank oder ein neues Modell erzeugen wollen, wir entscheiden uns für die erste Option:

Nachfolgend bestimmen Sie die zugrundeliegende Datenbank:

Nach dem Klick auf "Weiter" müssen Sie noch die per Modell abzubildenden Entitäten auswählen, wir belassen es bei der Tabelle "Products":



Nach kurzem Festplattenrasseln dürften Sie sich im EDM-Designer wiederfinden. Unser Modell sieht etwas karg aus, was aber nichts an der Funktionalität ändert:



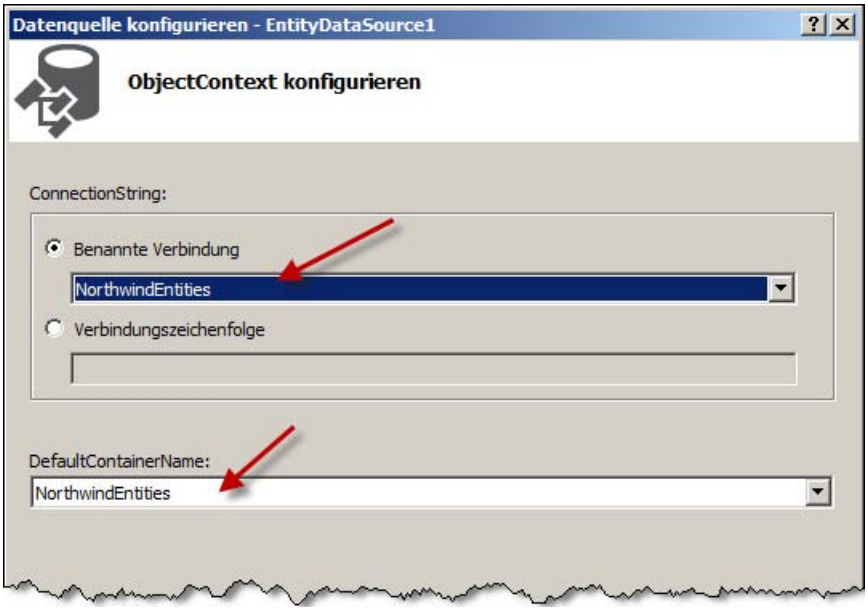
Fügen Sie mehr als eine Tabelle hinzu, erkennt der Assistent automatisch auch die Abhängigkeiten zwischen den Tabellen und erstellt die gewünschten Verbindungen.

Damit ist der Entwurf unseres Datenmodells abgeschlossen, und wir können uns um die Oberfläche und damit natürlich auch die *EntityDataSource* kümmern.

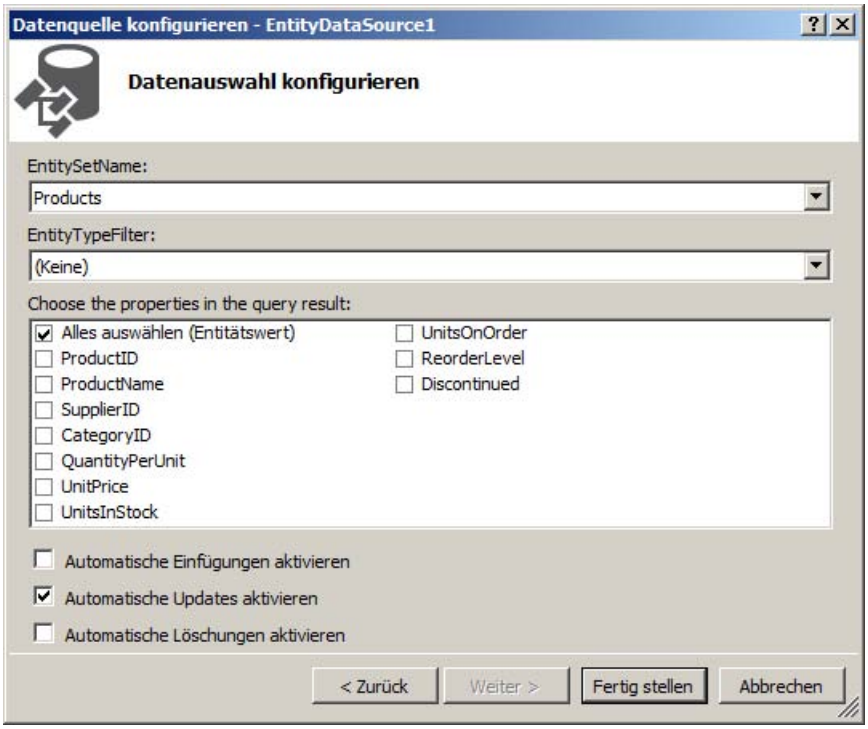
HINWEIS: Bevor Sie weiterarbeiten ist es sinnvoll, das Projekt einmal zu kompilieren, andernfalls kann es passieren, dass später einige Klassen nicht gefunden werden.

41.8.2 EntityDataSource anbinden

Fügen Sie dem Web-Formular eine *EntityDataSource* sowie ein *GridView* hinzu. Zunächst wollen wir die *EntityDataSource* konfigurieren. Klicken Sie dazu im Aufgabenmenü des Controls auf "Datenquelle konfigurieren" und wählen Sie, wie in der folgenden Abbildung gezeigt, unsere Verbindung "NorthwindEntities" aus.

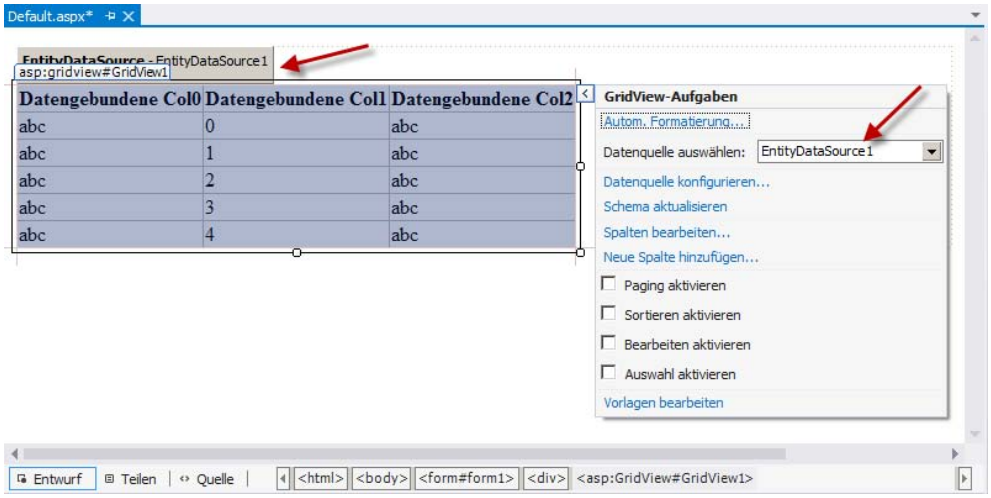


Nächster Schritt ist die Auswahl der gewünschten Entität, hier bleibt Ihnen keine große Wahl, wir haben nur *Products*:



HINWEIS: Einschränkungen bei den Spalten resultieren in einer Projektion (Abfrage).

Schließlich müssen wir noch die *GridView* an die *EntityDataSource* anbinden, auch dies gelingt dank Aufgabenmenü ohne große Probleme:



Nach Betätigen von F5 können Sie bereits die Tabellenansicht im Internet Explorer bewundern:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsIn
1	Chai	1	1	10 boxes x 20 bags	18,0000	39
2	Chang 777	1	1	24 - 12 oz bottles	19,0000	17
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10,0000	13
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000	53
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,3500	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25,0000	120
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30,0000	15
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40,0000	6
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97,0000	29
10	Ikura	4	8	12 - 200 ml jars	31,0000	31

41.8.3 Datenmenge filtern

Dass wir alle Daten angezeigt bekommen, ist im vorliegenden Fall nicht weiter schlimm, so umfangreich ist die Tabelle ja nicht. Ganz anders sieht die Thematik allerdings in realen Szenarien aus, hier ist fast immer mit dem Filtern der Daten zu rechnen. Doch wie bewerkstelligen wir dies im Zusammenhang mit der *EntityDataSource*?

Zwei Möglichkeiten bieten sich auf an:

- Sie filtern die Daten mit Hilfe der *Where*-Eigenschaft und verwenden dazu die entsprechenden ESQL[1]-Anweisungen.
- Sie nutzen den neuen *QueryExtender* (siehe Abschnitt 41.10 Seite 2015), der Sie nicht gleich mit einem neuen SQL-Dialekt "beglückt".

Ein kurzes Beispiel (alle Artikel mit "Chai") zeigt die Verwendung der ersten Variante:

BEISPIEL 41.9: Filtern mit der *Where*-Eigenschaft

C#

Da wir einen Bezug zum Datenmodell benötigen, müssen wir mit "it" zunächst auf dieses verweisen, um dann die eigentliche SQL-Anweisung aufzubauen:

```
<asp:EntityDataSource ID="EntityDataSource1" runat="server"
            ConnectionString="name=NorthwindEntities"
            DefaultContainerName="NorthwindEntities" EnableFlattening="False"
            EnableUpdate="True" EntitySetName="Products" EntityTypeFilter="" Select=""
            Where="it.ProductName = &quot;Chai&quot;">
        </asp:EntityDataSource>
```

Nachfolgend wird nur noch der eine passende Datensatz zurückgegeben:

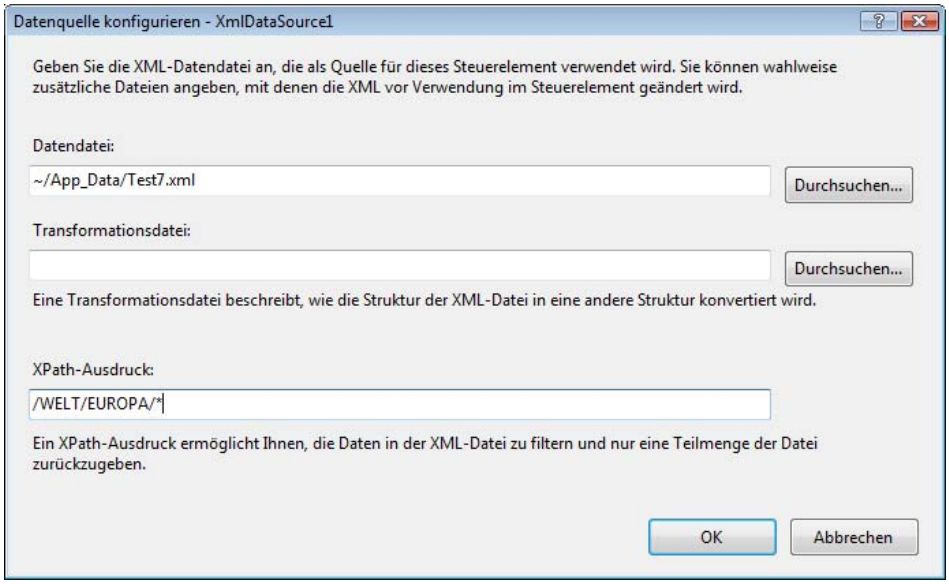
ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
1	Chai	1	1	10 boxes x 20 bags	18,0000	39	0

HINWEIS: Die Datenauswahl erfolgt dank LINQ to Entities auf dem Server, d.h., es wird auch nur ein Datensatz vom Server abgerufen, der Client muss sich nicht um das Filtern der Daten kümmern.

41.9 XmlDataSource

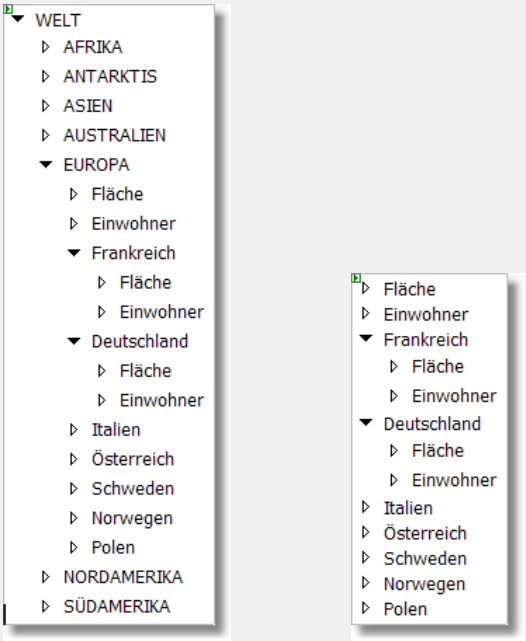
Als letzte Datenquelle wollen wir uns mit der *XmlDataSource* beschäftigen. Diese bietet den komfortablen Zugriff auf strukturierte XML-Daten, die Sie zum Beispiel im *App_Data*-Verzeichnis speichern können. Zusätzlich lassen sich diese Daten mittels Transformationsdatei (Eigenschaft *TransformFile* oder *Transform*) konvertieren oder mit einem XPath-Ausdruck filtern (Eigenschaft *XPath*):

[1] Entity SQL – eine vom physischen Datenmodell unabhängige Abfragesprache.



BEISPIEL 41.10: Verwendung von XPath

Aus dem kompletten XML-Baum (linke Abbildung) wird mittels obigem XPath-Ausdruck der Zweig EUROPA herausgefiltert (rechte Abbildung):



41.10 QueryExtender

Der *QueryExtender* ermöglicht es dem Programmierer, mit minimalem Aufwand umfangreiche Filter- und Sortierfunktionen zu den Datenquellen *LinqDataSource* und *EntityDataSource* hinzuzufügen. Ganz nebenbei können Sie bei beiden zugrundeliegenden Datenquellen die gleiche Syntax verwenden, es ist also später recht einfach möglich, statt einer *LinqDataSource* eine *EntityDataSource* einzusetzen, ohne die Logik des Formulars zu verändern.

41.10.1 Grundlagen

Doch wozu brauchen wir eigentlich den *QueryExtender*? Die Antwort ist schnell gegeben, wenn Sie versuchen, bestehenden Datenmodelle mit den obigen Datenquellen dahingehend zu nutzen, dass Sie beispielsweise einen Filter für bestimmte Spalten setzen wollen.

Wer jetzt an die Verwendung der *Where*-Eigenschaft bei beiden obigen Datenquellen denkt, liegt zwar nicht falsch, allerdings unterliegt diese Eigenschaft gewissen Beschränkungen und ist bei wechselnden Kriterien nicht flexibel genug. Ganz nebenbei ist auch die Syntax bei *LinqDataSource* und *EntityDataSource* verschieden, was es dem Programmierer bei einem späteren Wechsel nicht einfacher macht.

Genau hier ist der Ansatz des *QueryExtenders*, dieser soll eine einheitliche, einfache Möglichkeit bieten, sowohl *LinqDataSource*- als auch *EntityDataSource*-Daten zu filtern und zu sortieren. Dazu wird das Control zum einen an eine der obigen Datenquellen gebunden, zum anderen werden die Parameter für die Such-, Filter und/oder Sortierbefehle an entsprechende Eingabecontrols oder z.B. an Elemente des QueryStrings gebunden.

HINWEIS: Wer jetzt Sorge hat, dass dieses Konzept zu Lasten der Performance (Datendurchsatz) geht, liegt falsch, das Control beeinflusst direkt die an den Server gesendeten SQL-Anweisungen, es werden also nur die Daten geholt, die auch unbedingt nötig sind. Eine Nachverarbeitung auf dem Client findet nicht statt.

Die folgende Tabelle listet kurz die Möglichkeiten auf, die Ihnen der *QueryExtender* bietet:

Ausdruck	Bemerkung
SearchExpession	Entspricht der altbekannten *Where*-Klausel, d h. der Suche in einem Feld und dem Vergleich mit einer Zeichenkette.
RangeExpression	Der gesuchte Wert muss sich zwischen einem vorgegebenen Minimum und einem Maximum bewegen.
PropertyExpression	Filtern basierend auf Eigenschaften.
MethodExpression	Filtern mit Hilfe einer selbst zu definierenden Methode.
OrderByExpression	Sortieren von Daten mit Vorgabe der Sortierrichtung.
CustomExpression	Mit Hilfe einer aufzurufenden Methode können beliebige user-spezifische Filterausdrücke erstellt werden

Gleich noch ein wichtiger Hinweis vorweg:

HINWEIS: Das Control können Sie zwar noch per *TagetControlId*-Eigenschaft an eine DataSource anbinden, danach müssen Sie Ihre Nase aber tief in die Quellcodeansicht der Webseite stecken, um obige Ausdrücke zu erstellen, ohne XML-Code geht hier gar nichts.

41.10.2 Suchen

Erstellen Sie zunächst ein Datenmodell (LINQ to SQL oder EDM) zum Beispiel für die *Northwind*-Datenbank und binden Sie eine *LinqDataSource* oder *EntityDataSource* an dieses Modell (Tabelle *Products*). Nachfolgend können Sie zum Beispiel ein *GridView* an diese Datenquelle binden, um sich von der Funktionsweise zu überzeugen. Nach dem Start sollten in jedem Fall zunächst alle Daten der Tabelle angezeigt werden.

In einem zweiten Schritt fügen Sie jetzt einen *QueryExtender* hinzu und binden diesen über die *TargetControlID* an die Datenquelle.

Für die Suche benötigen wir auch ein Suchkriterium, wir nutzen eine *TextBox* zur Eingabe und einen Schaltfläche für den Postback der Seite (erst dann werden auch die Daten gefiltert).

BEISPIEL 41.11: Filtern nach Produktnamen

HTML

Die Anbindung an das EDM per *EntityDataSource*-Control realisieren:

```
<asp:EntityDataSource ID="EntityDataSource1" runat="server"
    ConnectionString="name=NorthwindEntities"
    DefaultContainerName="NorthwindEntities" EnableFlattening="False"
    EntitySetName="Products">
</asp:EntityDataSource>
```

Hier die Parametrierung des *QueryExtenders*:

```
<asp:QueryExtender ID="QueryExtender1" TargetControlID="EntityDataSource1"
                   runat="server">
```

Wir haben einen *SearchExpression,* der auf das Feld *ProductName* angewendet wird, Quelle ist die *TextBox1*:

```
    <asp:SearchExpression DataFields="ProductName" SearchType="StartsWith">
      <asp:ControlParameter ControlID="TextBox1" />
    </asp:SearchExpression>
</asp:QueryExtender>
```

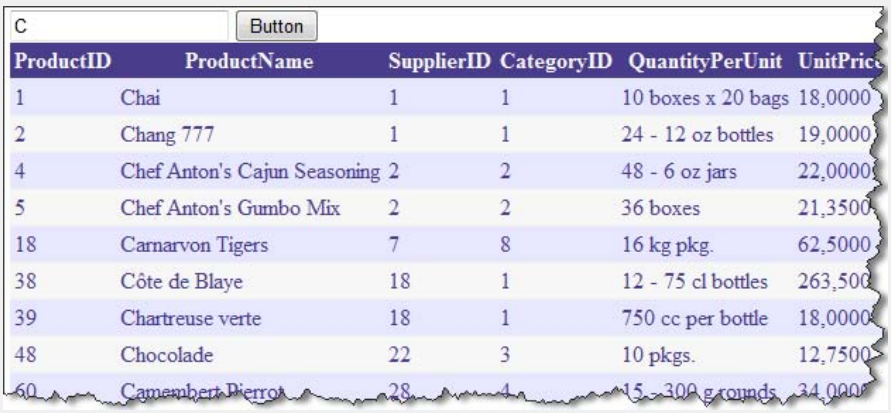
Hier noch die Definition der *TextBox* und der Schaltfläche:

```
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:Button ID="Button1" runat="server" Text="Button" />
<br />
```

BEISPIEL 41.11: Filtern nach Produktnamen

Ergebnis

Nach dem Start werden zunächst alle Einträge angezeigt, geben Sie jetzt beispielsweise ein "C" in die *TextBox* ein und klicken auf die Schaltfläche, so wird ein Postback zum Server ausgelöst und wir erhalten die gefilterte Datenmenge:



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPric
1	Chai	1	1	10 boxes x 20 bags	18,0000
2	Chang 777	1	1	24 - 12 oz bottles	19,0000
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22,0000
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,3500
18	Carnarvon Tigers	7	8	16 kg pkg.	62,5000
38	Côte de Blaye	18	1	12 - 75 cl bottles	263,500
39	Chartreuse verte	18	1	750 cc per bottle	18,0000
48	Chocolade	22	3	10 pkgs.	12,7500
60	Camembert Pierrot	28	4	15 - 300 g rounds	34,0000

Bei einer Bereichssuche mittels *RangeExpression* werden logischerweise auch zwei Vorgabewerte (für Min und Max) erwartet, hier können Sie beispielsweise zwei *TextBox*en anbinden. Ob der Bereich die Grenzen mit einschließt, können Sie über die Attribute *MinType/MaxType* (*Inclusive, Exclusive, None*) festlegen.

BEISPIEL 41.12: Filtern mit Bereich

HTML

Verwenden Sie in diesem Fall einen *RangeExpression*:

```
<asp:QueryExtender ID="QueryExtender1" TargetControlID="EntityDataSource1" runat="server">
            <asp:SearchExpression DataFields="ProductName" SearchType="StartsWith">
              <asp:ControlParameter ControlID="TextBox1" />
            </asp:SearchExpression>
```

Bestimmen Sie zunächst das Datenfeld (*DataField*) sowie die Bereichsgrenzen (*MinTyp, MaxType*):

```
            <asp:RangeExpression DataField="UnitPrice" MinType="Inclusive"
                                 MaxType="Inclusive">
```

Hier werden die Werte zweier *TextBox*en für Min und Max verwendet:

```
              <asp:ControlParameter ControlID="TextBox2" />
              <asp:ControlParameter ControlID="TextBox3" />
            </asp:RangeExpression>
        </asp:QueryExtender>
```

Möchten Sie nach Eigenschaften filtern, bietet sich ein *PropertyExpression* an, wie es das folgende Beispiel zeigt:

BEISPIEL 41.13: Filtern nach Eigenschaft

XAML

In Abhängigkeit von der *CheckBox1* sollen nur Produkte angezeigt werden, welche die Option "discontinued" haben:

```
<asp:QueryExtender ID="QueryExtender1" TargetControlID="EntityDataSource1" runat="server">
   <asp:PropertyExpression>
      <asp:ControlParameter Name="Discontinued" ControlID="CheckBox1" />
   </asp:PropertyExpression>
</asp:QueryExtender>
```

Last but not least, können Sie auch Ihre C#-Kenntnisse einbringen, indem Sie eine eigene Filter-Methode definieren, die Sie per *CustomExpression* einbinden:

BEISPIEL 41.14: Filtern mit Ereignis

HTML

```
<asp:QueryExtender ID="QueryExtender1" TargetControlID="EntityDataSource1" runat="server">
```

Hier verweisen Sie auf die C#-Methode:

```
            <asp:CustomExpression OnQuerying="MeinFilter" />
        </asp:QueryExtender>
```

C#

```
...
using System.Web.UI.WebControls.Expressions;

public partial class Bsp_QueryExtender : System.Web.UI.Page
{
    protected void MeinFilter(object sender, CustomExpressionEventArgs e)
    {
        e.Query = from p in e.Query.Cast<NorthwindModel.Products>()
                  where p.ProductName.Contains("oco") select p;
    }
```

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock
19	Teatime Chocolate Biscuits	8	3	10 boxes x 12 pieces	9,2000	25
48	Chocolade	22	3	10 pkgs.	12,7500	15

41.10.3 Sortieren

Dass auch das Sortieren mit Hilfe des *QueryExtender*-Controls kein Problem ist, zeigt das folgende kurze Beispiel:

BEISPIEL 41.15: Sortieren nach der Spalte *ProductName*

HTML

```
        <asp:QueryExtender ID="QueryExtender1" TargetControlID="EntityDataSource1"
                           runat="server">
            <asp:OrderByExpression DataField="ProductName" Direction="Descending" />
        </asp:QueryExtender>
```

41.11 GridView

Hier haben wir es mit dem direkten Nachfolger des "berühmt berüchtigten" *DataGrid* zu tun. "Berüchtigt" für die doch recht aufwändige Programmierung selbst einfachster Aufgaben (Sortieren, Paging etc.), "berühmt" für die Fortschritte gegenüber der alten ASP-Programmierung.

Vergessen Sie all Ihre schlechten Erfahrungen und lassen Sie sich von der langen Feature-Liste begeistern:

- absolut einfache Bindung an die neuen Datenquellen per *DataSourceId*,
- Funktion zum Auswählen von Zeilen,
- Paging-Funktion,
- Sortier-Funktion,
- Edit-, Update-, Delete-Funktion,
- Unterstützung für Bilder (*ImageField*) und bool'sche Werte (*CheckBoxField*),
- Hyperlink-Spalten für Detailanzeigen etc.,
- diverse Formatierungsmöglichkeiten mit Themes und Styles.

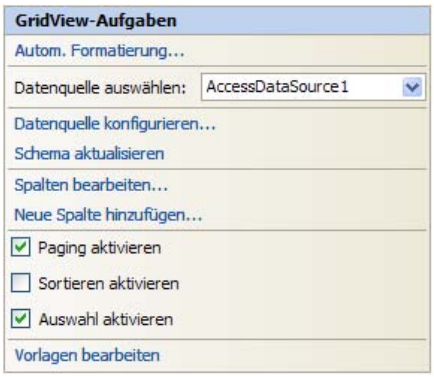
Id	Bezeichnung	Preis	Anzahl			
1	lange Hosen	45	5555	Bearbeiten	Löschen	Auswählen
3	Blaue Jacken	45	57	Bearbeiten	Löschen	Auswählen
4	lange Hosen	45	5555	Bearbeiten	Löschen	Auswählen
5	Hüte	123	5	Bearbeiten	Löschen	Auswählen
6	Gurken	333	2	Bearbeiten	Löschen	Auswählen

Einige thematische Beispiele demonstrieren Ihnen die Verwendung, die endlose Aufzählung von Eigenschaften und Methoden ersparen wir Ihnen besser.

41.11.1 Auswahlfunktion (Zeilenauswahl)

Eine der wichtigsten Funktionen neben der reinen Anzeige von Tabellendaten ist sicher auch die Auswahl eines Eintrags, um diesen zum Beispiel zu editieren oder auch um Detaildaten anzuzeigen.

Aktivieren Sie dazu über das Aufgaben-Menü die Option *Auswahl aktivieren*:

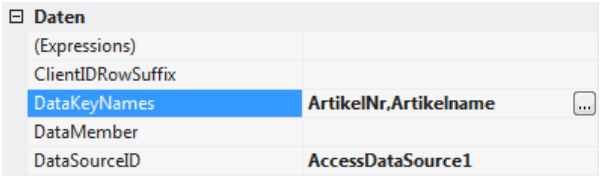


Setzen Sie jetzt noch die Eigenschaft *DataKeyNames* auf den Primärschlüssel der Tabelle und Sie können im Ereignis *SelectedIndexChanged* die gewünschte Information abrufen:

```
protected void GridView1_SelectedIndexChanged(object sender, EventArgs e)
{
    Label1.Text = GridView1.SelectedValue.ToString();
}
```

41.11.2 Auswahl mit mehrspaltigem Index

Sollten Sie mehr als eine Spalte als Primärschlüssel festgelegt haben (Eigenschaft *DataKeyNames* mit kommagetrennter Liste der Spaltennamen),



... können Sie auch auf jede einzelne Spalte zugreifen:

```
protected void GridView1_SelectedIndexChanged(object sender, EventArgs e)
{
    Label2.Text = GridView1.SelectedDataKey.Value.ToString();
    Label3.Text = GridView1.SelectedDataKey.Values[1].ToString();
}
```

41.11.3 Hyperlink-Spalte für Detailansicht

Möchten Sie eine neue Seite mit den Detaildaten der aktuellen Auswahl anzeigen, können Sie, wie im vorhergehenden Abschnitt demonstriert, zunächst den Primärschlüssel ermitteln und diesen dann per *Session* oder *QueryString* an die Detailseite weiterreichen. Doch wozu soviel Arbeit, wenn es auch automatisiert geht?

Fügen Sie in das *GridView* eine zusätzliche Spalte "Details" vom Typ *HyperlinkField* ein. Die Eigenschaft *DataNavigateUrlField* legen Sie auf den Primärindex der Tabelle fest. Über *DataNavigateUrlFormatString* können Sie den Url mit Hilfe von Platzhaltern ({0}, {1} ...) zusammenbauen:

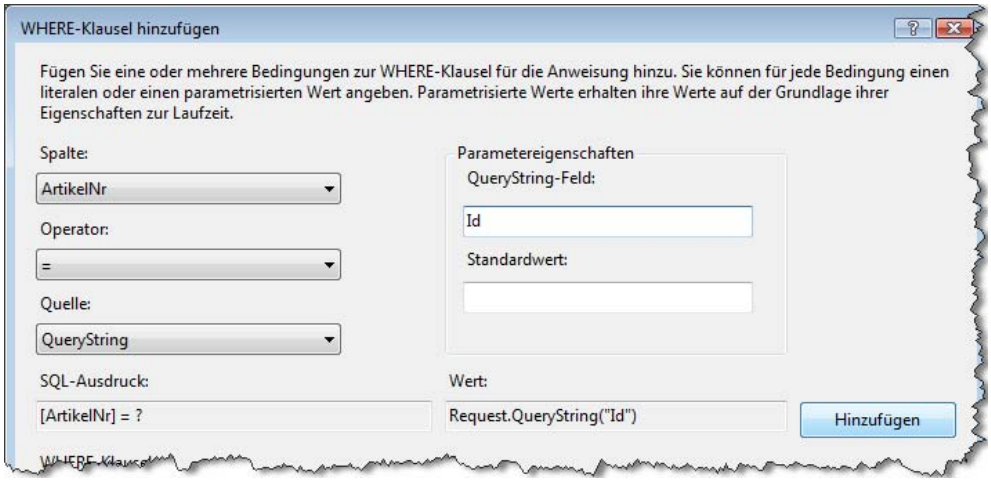
```
<asp:GridView ID="GridView1" runat="server" AllowPaging="True" AutoGenerateColumns="False"
...
   <asp:HyperLinkField DataNavigateUrlFields="ArtikelNr"
                       DataNavigateUrlFormatString="~/Detaildaten.aspx?Id={0}"
                       Text="Details" />
...
</asp:GridView>
```

Damit ist der Entwurf für das *GridView* abgeschlossen. Die Detailseite braucht sich jetzt nur noch um die Auswertung des QueryStrings zu kümmern.

Dies kann entweder per Code erfolgen

```
    protected void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = Request.QueryString["Id"];
    }
```

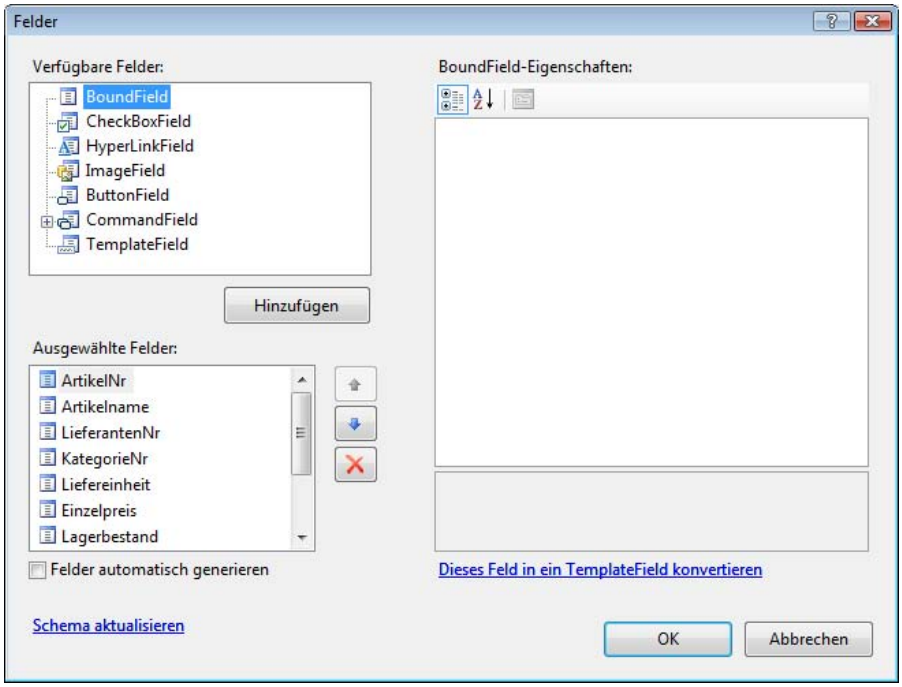
... oder durch Zuweisung eines WHERE-Parameters für die Detaildatenauswahl:



41.11.4 Spalten erzeugen

Neben der bereits bekannten Variante, bei der die Spalten automatisch generiert wurden, können Sie diese auch selbst erzeugen. Nutzen Sie dazu das Aufgaben-Menü (*Spalten bearbeiten* bzw. *Neue Spalte hinzufügen*).

Im folgenden Editor schalten Sie zunächst die Option *Felder automatisch generieren* ab (siehe folgende Abbildung).



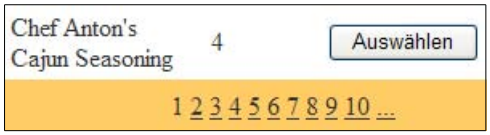
Wählen Sie einen Feldtyp aus der linken oberen Liste aus und fügen Sie diesen ein. Die verfügbaren Eigenschaften werden Ihnen nachfolgend in der rechten Liste angezeigt. Die wichtigste Eigenschaft dürfte *DataField* sein, der Name der anzuzeigenden Spalte.

Zu den schon vom *DataGrid* her bekannten Spaltentypen kommen jetzt noch *CheckBox*-Felder (für *True*/*False*-Werte) und *Image*-Felder hinzu.

Auch Spalten mit zusätzlichen Schaltflächen (*ButtonField*) sind neben der *CommandField*-Spalte (Edit, Update, Delete etc.) realisierbar. Damit lassen sich beispielsweise per Ereigniscode Markierungen setzen oder Detaildaten anzeigen.

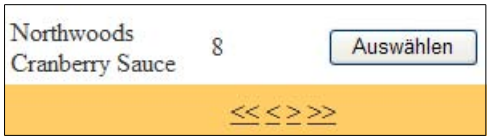
41.11.5 Paging realisieren

Grundsätzlich genügt es, wenn Sie die Eigenschaft *AllowPaging* auf *True* setzen, um die komplette Paging-Funktionalität zu aktivieren:



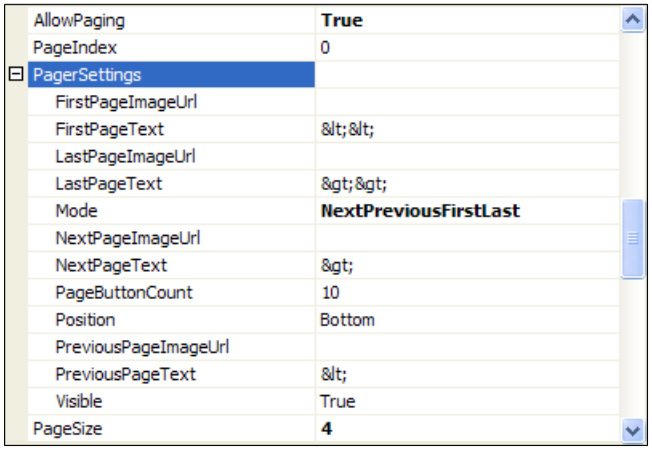
Zusätzlich können Sie mit *PageSize* die Anzahl der gleichzeitig angezeigten Datensätze beeinflussen.

Statt der Nummernliste können Sie auch Navigationsschaltflächen einblenden (Next, Previous, First, Last).

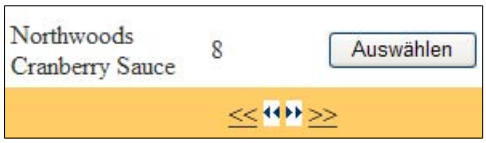


Ob, und wenn ja welche Schaltflächen, bestimmt die Eigenschaft *Mode*.

Dem Ästheten sind die textuellen Schaltflächen sicher ein Grauen. Abhilfe schaffen die *PagerSettings*-Eigenschaften, über die Sie auch Grafiken einblenden können:



NextPageImageUrl und *PreviousPageImageUrl* wurden festgelegt

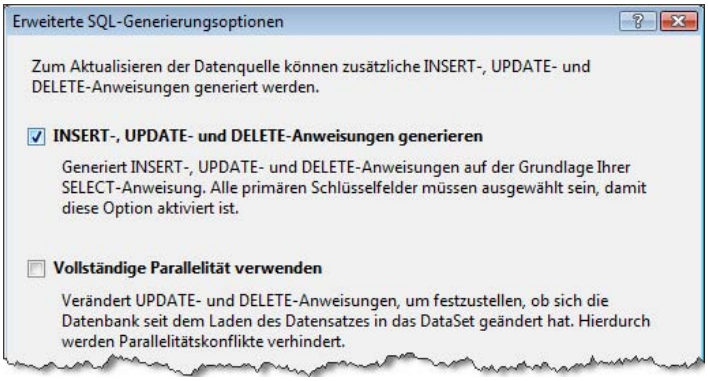


HINWEIS: Last but not least, können Sie mit *Position* festlegen, ob der Pagerbereich oben oder unten angezeigt wird.

41.11.6 Edit, Update, Delete

Sollen nicht nur statische Daten angezeigt werden, muss Ihre Datenquelle sinnvollerweise auch über UPDATE und DELETE-Anweisungen verfügen, anderenfalls dürften Sie sich vergeblich um die entsprechende Funktionalität bemühen.

Die entsprechenden Anweisungen generieren Sie am besten mit dem DataSource-Assistenten, nachdem Sie das entsprechende SELECT-Statement erzeugt haben (siehe folgende Abbildung).



Die entsprechenden Aktionen können Sie zur Laufzeit über die Schaltflächen des *CommandFields* auslösen.

Alternativ kann auch mit der Eigenschaft *EditIndex* die jeweilige **Zeile** in den Editiermodus versetzt werden, mit *EditIndex = -1* schalten Sie in den normalen Ansichtsmodus zurück.

HINWEIS: Hier wird mit *GridView*-Zeilen (0 ... n), nicht mit Tabellenzeilen, gearbeitet!

Analoges gilt für die Auswahl einer angezeigten Zeile: Über *SelectedIndex* aktivieren Sie die gewünschte Zeile.

HINWEIS: Das Ereignis *SelectedIndexChanged* wird in diesem Fall nicht ausgelöst!

41.11.7 Keine Daten, was tun?

Enthält die zugrunde liegende Tabelle/Abfrage keine Daten, wird das *GridView* nicht angezeigt. Sinnvollerweise sollten Sie für diesen Fall die Eigenschaft *EmptyDataText* festlegen, statt des *GridView* wird dann dieser Text angezeigt.

41.12 DetailsView

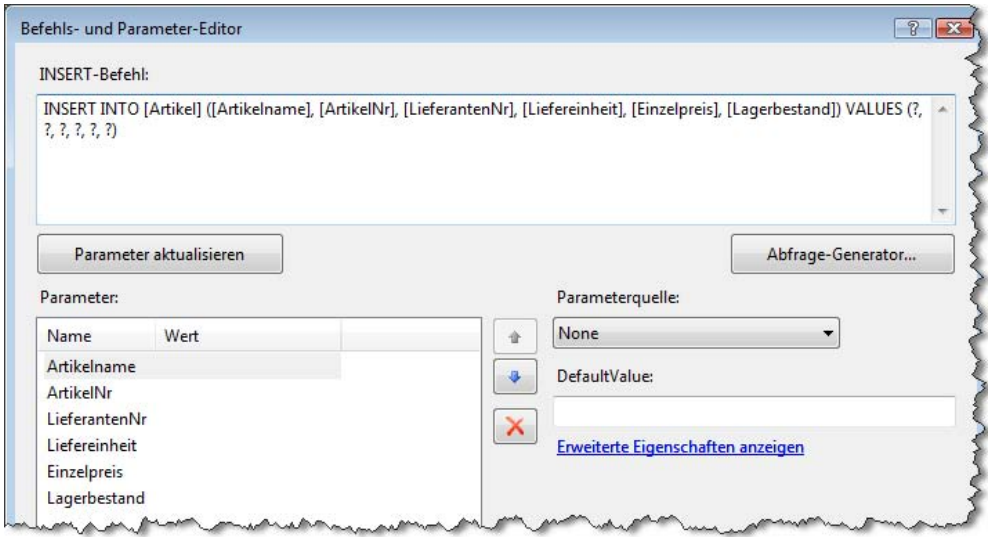
Im Gegensatz zum *GridView* wird durch das *DetailsView* nur ein Datensatz angezeigt. Auch hier ist das Editieren, Löschen und vor allem auch das Einfügen von Datensätzen möglich.

ArtikelNr	2
Artikelname	Chang
LieferantenNr	1
KategorieNr	1
Liefereinheit	24 x 12-oz-Flaschen
Einzelpreis	19
Lagerbestand	17
BestellteEinheiten	40
Mindestbestand	25
Auslaufartikel	☐

Bearbeiten Löschen Neu

Haben Sie zusätzlich Paging aktiviert, steht auch dem Navigieren durch die Datenmenge nichts mehr im Weg.

Neben der Anzeige von Detaildaten dürfte vor allem das Einfügen neuer Datensätze zu den Hauptaufgaben des Steuerelements zählen. Dazu muss die zugrunde liegende Datenquelle einen INSERT-Befehl realisieren:



Zusätzlich müssen Sie den Insert-Button über das Aufgaben-Menü einblenden, alternativ können Sie auch die Eigenschaft *AutoGenerateInsertButton* auf *True* setzen.

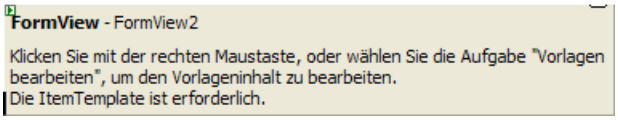
Welche Spalten editierbar sind, bestimmen Sie mit dem Feldeditor. Nutzen Sie die Eigenschaft *InsertVisible* (*True*/*False*) um Spalten gezielt auszublenden. Über die Eigenschaft *DefaultMode* können Sie zwischen *ReadOnly*, *Edit* und *Insert* wechseln, d h., beim ersten Aufruf wird das Steuerelement in der gewünschten Ansicht dargestellt. Zur Laufzeit können Sie mit der Methode *ChangeMode* zwischen den verschiedenen Anzeigemodi wechseln.

BEISPIEL 41.16: Insert- und Edit-Mode

Die allgemeine Verwendung/Formatierung entspricht weitgehend dem *GridView* (siehe vorhergehender Abschnitt).

41.13 FormView

Wesentlich leistungsfähiger, dafür aber auch aufwändiger zu programmieren als ein *DetailsView*-Control, ist das *FormView*. Es bietet ebenfalls die Möglichkeit, einen Datensatz anzuzeigen bzw. zu editieren, schränkt den Programmierer aber nicht auf ein festes Layout ein, d.h., Sie können den Inhalt des *FormView* mit den zur Verfügung stehenden Server-Controls (*Label*, *TextBox* etc.) frei gestalten. Dabei zeigt sich das Control zunächst von seiner "grauen" Seite:



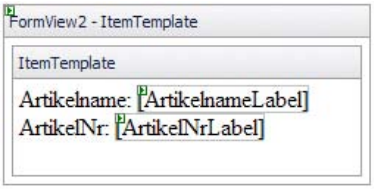
Ganz so schnell wie mit den anderen Controls kommen Sie hier aber nicht ans Ziel. Dazu müssen Sie zumindest ein *ItemTemplate* erzeugen. Alternativ stehen Ihnen noch folgende Templates zur Verfügung:

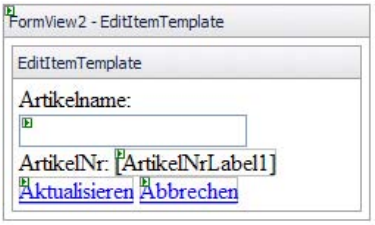
Template	Beschreibung
EditItemTemplate	Controls/Aussehen für den Editiermodus
EmptyDataTemplate	Anzeige bei leerer Datenmenge
FooterTemplate	Aussehen des Fuß-Bereichs
HeaderTemplate	Aussehen des Kopf-Bereichs
InsertItemTemplate	Controls/Aussehen für den Einfügemodus

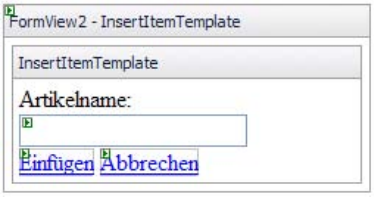
Template	Beschreibung
ItemTemplate	Controls/Aussehen für den normalen Anzeigemodus
PagerTemplate	Aussehen des Pager-Bereichs

Die einzelnen Templates können Sie über das Aufgaben-Menü des Controls (*Vorlagen bearbeiten*) editieren:

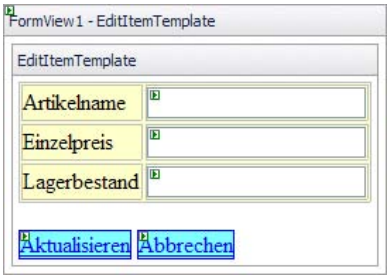
Haben Sie vorher die Datenquelle zugewiesen, werden automatisch ein *ItemTemplate*, ein *EditItemTemplate* und ein *InsertItemTemplate* erzeugt:







Jedes diese Templates können und werden Sie aus optischen Gründen sicher auch editieren. Positionieren Sie die einzelnen Steuerelemente am besten mit Hilfe von HTML-Tabellen, wie im Folgenden gezeigt:



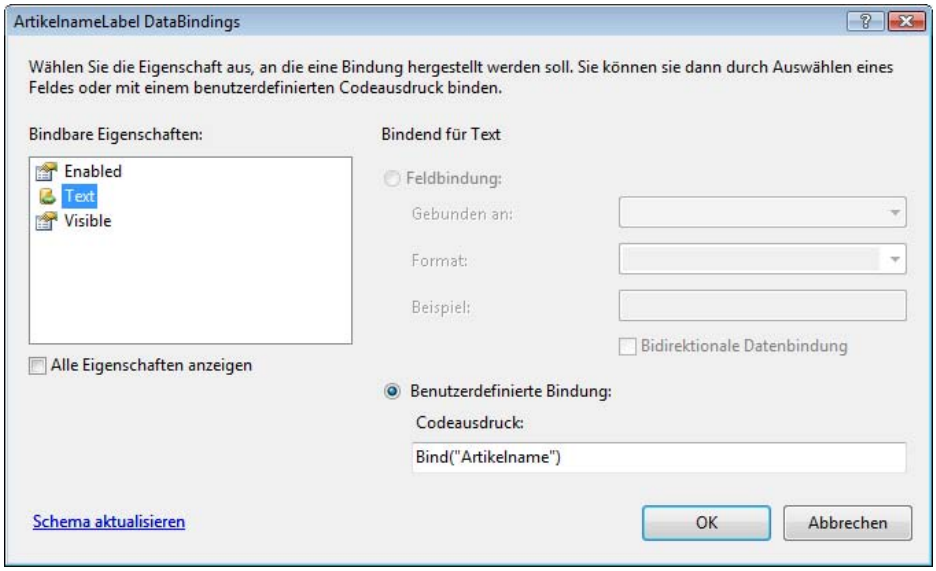
Wem die automatisch erzeugten Schaltflächen nicht gefallen, der kann auch ganz normale Buttons verwenden, allerdings müssen Sie deren *CommandName*-Eigenschaft mit *Insert*, *Edit*, *Update* oder *Cancel* festlegen, um dem Button auch die gewünschte Funktionalität zu geben.

Das grundsätzliche Verhalten des Steuerelements legen Sie, wie auch beim *DetailsView*-Control, mit der Eigenschaft *DefaultMode* (*ReadOnly*, *Edit*, *Insert*) oder zur Laufzeit mit der Methode *ChangeMode* fest.

Datenbindung im Detail

Haben Sie das Kapitel bis hierher aufmerksam studiert, werden Sie sich vielleicht verwundert die Augen reiben, lassen sich doch plötzlich viele Steuerelemente, bei denen das vorher nicht möglich war, an Daten binden. Dies ist in der Tat so, da sich die Controls nun in einem Template befinden.

Sehen wir uns zum Beispiel eine *TextBox* näher an, stellen wir erstaunt fest, dass diese plötzlich auch über ein Aufgabenmenü für Datenbindung verfügt, das unsere Wünsche voll und ganz erfüllen kann:



Im ASP-Quellcode liest sich das dann so:

```
<asp:FormView ID="FormView1" runat="server" DataKeyNames="ArtikelNr"
              DataSourceID="AccessDataSource2"
              DefaultMode="Edit" OnPageIndexChanging="FormView1_PageIndexChanging">
  <EditItemTemplate>
...
  <asp:TextBox ID="LagerbestandTextBox" runat="server"
```

```
            Text='<%# Bind("Artikelname") %>'></asp:TextBox></td>
...
</asp:FormView>
```

Mit *Bind(...)* wird eine Zweiwege-Datenbindung erzeugt (Anzeigen und Schreiben), alternativ kann auch *Eval(...)* zum Einsatz kommen, in diesem Fall beschränkt sich die Funktion auf die pure Anzeige.

Neben der reinen Auswahl eines Tabellenfelds kann an dieser Stelle auch gleich eine Formatierung vorgenommen werden, übergeben Sie dazu der *Eval*- bzw. der *Bind*-Funktion als zweiten Parameter einen Formatierungsstring.

BEISPIEL 41.17: Währungsformatierung

XML

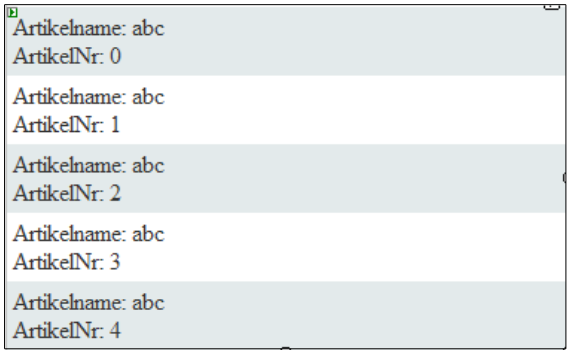
```
... Bind("Einzelpreis", "{0:C}") ...
```

Ergebnis

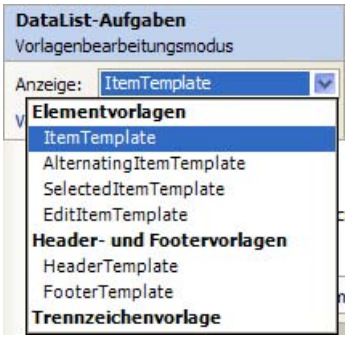
Artikelname	Chang
Einzelpreis	19,00 €
Lagerbestand	17

41.14 DataList

Auch das *DataList*-Control basiert wie das *FormView* auf Templates, die in diesem Fall jedoch in Listenform für alle Datensätze abgearbeitet werden.



Wie auch beim *GridView* können Sie mit diesem Steuerelement Datensätze selektieren, editieren und löschen. Daraus können Sie nach den bisherigen Erfahrungen schon ableiten, dass es auch in diesem Fall mehrere Templates gibt, die einzeln zu programmieren sind (siehe folgende Abbildung).



41.14.1 Bearbeitungsfunktionen implementieren

Die Vorgehensweise ist in diesem Fall leider eine Mischung aus alten und neuen Technologien. Zum Ändern des Bearbeitungsmodus müssen Sie in die jeweiligen Templates Schaltflächen einfügen, einen der folgenden Befehle implementieren (*CommandName*-Eigenschaft):

- *edit*
- *update*
- *cancel*
- *delete*
- *select*

HINWEIS: Achten Sie darauf, in welches Template Sie welches Command einfügen.

Damit ist es allerdings noch nicht getan, lösen doch diese Zuordnungen "lediglich" die entsprechenden Ereignisse (*EditCommand* ...) aus. Mit diesen Ereignissen müssen Sie dann die eigentlichen Aktionen anstoßen.

BEISPIEL 41.18: Edit-Modus setzen

C#
```
protected void DataList1_EditCommand(object source, DataListCommandEventArgs e)
{
    DataList1.EditItemIndex = e.Item.ItemIndex;
    DataList1.DataBind();
}
```

BEISPIEL 41.19: Cancel auslösen

C#
```
protected void DataList1_CancelCommand(object source, DataListCommandEventArgs e)
{
    DataList1.EditItemIndex = -1;
    DataList1.DataBind();
}
```

BEISPIEL 41.20: Update auslösen

```
C#
protected void DataList1_UpdateCommand(object source, DataListCommandEventArgs e)
{
  SqlDataSource1.UpdateParameters["Artikelnr"].DefaultValue =
        DataList1.DataKeys[e.Item.ItemIndex].ToString();
  SqlDataSource1.UpdateParameters["Artikelname"].DefaultValue =
        ((TextBox)e.Item.FindControl("TextBox1")).Text;
  SqlDataSource1.Update();
  DataList1.EditItemIndex = -1;
  DataList1.DataBind();
}
```

BEISPIEL 41.21: Delete auslösen

```
C#
protected void DataList1_DeleteCommand(object source, DataListCommandEventArgs e)
{
  SqlDataSource1.DeleteParameters["Artikelnr"].DefaultValue =
         DataList1.DataKeys[e.Item.ItemIndex].ToString();
  SqlDataSource1.Delete();
  DataList1.DataBind();
}
```

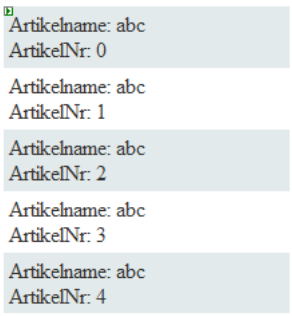
BEISPIEL 41.22: Select auslösen

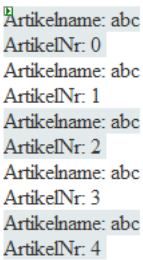
```
C#
protected void DataList1_SelectedIndexChanged(object sender, EventArgs e)
{
   DataList1.DataBind();
}
```

Möglicherweise überkommen Sie jetzt zwiespältige Erinnerungen an die Programmierung des *DataGrid*, aber anders geht es leider nicht.

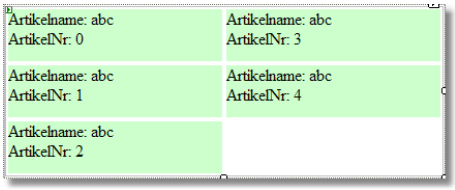
41.14.2 Layout verändern

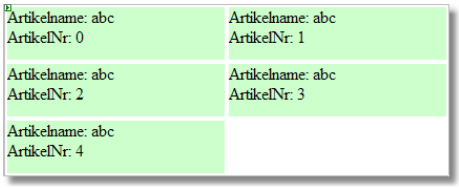
Sie können zwischen einem *Table*-Layout (linke Abbildung, Verwendung von HTML-Tabellen) und einem *Flow*-Layout (rechte Abbildung) über die Eigenschaft *RepeatLayout* wählen:





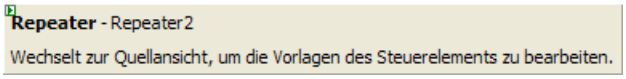
Mit *RepeatColumns* bestimmen Sie, wie oft das Template in horizontaler Richtung wiederholt wird (Spalten), *RepeatDirection* bestimmt die Reihenfolge in der die Datensätze zugewiesen werden (linke Abbildung *Vertical*, rechte Abbildung *Horizontal*).





41.15 Repeater

Auch bei diesem Control werden Sie zunächst mit einem der schönen grauen Kästen konfrontiert.



"Zunächst" ist eigentlich falsch ausgedrückt, denn es bleibt im Wesentlichen dabei, eine visuelle Unterstützung haben Sie bei diesen Control, das ein Template für alle vorhandenen Datensätze abarbeitet, leider nicht. Doch keine Sorge, mit einigen HTML-Kenntnissen und dem Wissen des vorhergehenden Kapitels kommen Sie auch hier weiter.

BEISPIEL 41.23: Repeater

HTML

Wechseln Sie also, wie im Control angegeben, in die Quellcode-Ansicht (HTML-Code) und geben Sie folgende Elemente ein, um eine einfache Artikelliste auszugeben:

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="AccessDataSource1">
   <ItemTemplate>
     Artikel: <%#Eval("Artikelname")%>
   </ItemTemplate>
</asp:Repeater>
```

Ergebnis

Das Resultat ist folgender Datenwust:

Artikel: Chai Artikel: Chang Artikel: Aniseed Syrup Artikel: Chef Anton's Cajun Seasoning Artikel: Chef Anton's Gumbo Mix Artikel: Grandma's Boysenberry Spread Artikel: Uncle Bob's Organic Dried Pears Artikel: Northwoods Cranberry Sauce Artikel: Mishi Kobe Niku Artikel: Ikura Artikel: Queso Cabrales Artikel: Queso Manchego La Pastora Artikel: Konbu Artikel: Tofu Artikel: Genen Shouyu Artikel: Pavlova

Wie Sie sehen, müssen Sie sich jetzt um alles selbst kümmern, dies betrifft auch die Zeilenumbrüche, Textformatierungen etc.

BEISPIEL 41.24: Repeater 2. Versuch

HTML

Mit etwas gutem Willen können wir aber schnell noch einige Verbesserungen realisieren (Fettschrift, Leerzeichen und Zeilenumbruch):

```
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="AccessDataSource1">
   <ItemTemplate>
      <b>Artikel:</b> <%#Eval("Artikelname")%><br/>
   </ItemTemplate>
</asp:Repeater>
```

Ergebnis

Und das sollte dabei rauskommen:

Artikel: Chai
Artikel: Chang
Artikel: Aniseed Syrup
Artikel: Chef Anton's Cajun Seasoning
Artikel: Chef Anton's Gumbo Mix
Artikel: Grandma's Boysenberry Spread
Artikel: Uncle Bob's Organic Dried Pears
Artikel: Northwoods Cranberry Sauce
Artikel: Mishi Kobe Niku

An dieser Stelle wollen wir aber den Ausflug in die Untiefen der HTML-Programmierung lieber abbrechen und stattdessen auf die zahlreiche Literatur zum Thema verweisen.

41.16 ListView

Dieses Control ist schlichtweg die "eierlegende Wollmilchsau" des ASP.NET-Programmierers. Neben der Möglichkeit, Daten in Listenform per Templates auszugeben (wie auch bei *DataList* und *Repeater),* bietet dieses Control zusätzlich diverse Konfigurationsmöglichkeiten, von denen zunächst die Unterstützung für das Einfügen, Löschen und Editieren sowie das Sortieren ins Auge stechen. Über ein *DataPager*-Control können Sie zusätzlich Einfluss auf das Paging nehmen. Verschiedene Templates (*LayoutTemplate, ItemTemplate, GroupTemplate, EmptyItemTemplate, SelectedItemTemplate* ...) ermöglichen es dem Programmierer das Control an fast jede Anforderung anzupassen.

HINWEIS: Mehr dazu in unserem [Visual C# 2012 Kochbuch].

41.17 Typisierte Datenbindung

Eine recht interessante Neuerung von ASP.NET ist die Möglichkeit, eine streng typisierte Datenbindung zu realisieren, indem den Daten-Controls der genutzte Typ zugewiesen wird. Ein Beispiel zeigt die Verwendung bei einer *DataList*, im Zusammenhang mit einem Entity-Data-Model:

BEISPIEL 41.25: Typisierte Datenbindung

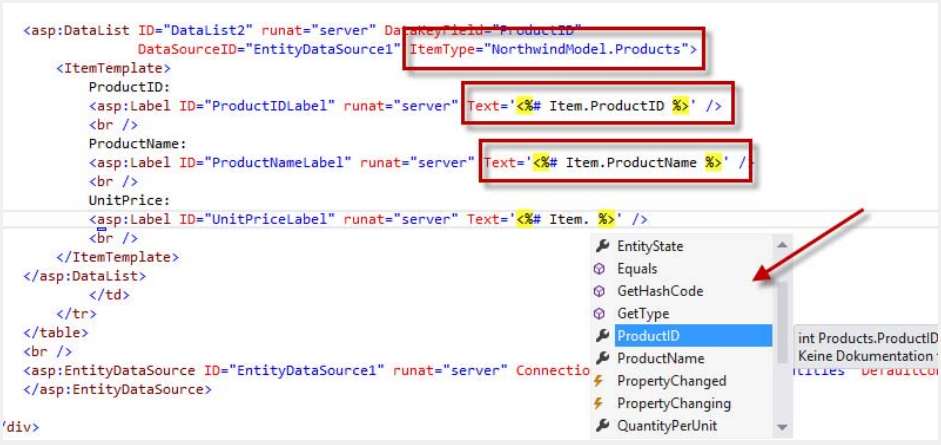
HTML

Lösung alt (vor ASP.NET 4.5):

```
...
 <asp:DataList ID="DataList1" runat="server" DataKeyField="ProductID"
               DataSourceID="EntityDataSource1">
     <ItemTemplate>
       ProductID:
       <asp:Label ID="ProductIDLabel" runat="server" Text='<%# Eval("ProductID") %>' />
       <br />
       ProductName:
       <asp:Label ID="ProductNameLabel" runat="server" Text='<%# Eval("ProductName") %>' />
       <br />
       UnitPrice:
       <asp:Label ID="UnitPriceLabel" runat="server" Text='<%# Eval("UnitPrice") %>' />
       <br />
     </ItemTemplate>
 </asp:DataList>
...
```

HTML

Lösung neu (mit ASP.NET 4.5):



Wie Sie sehen, wird über das Attribut *ItemType* zunächst der gewünschte Datentyp (in diesem Fall aus unserem Entity Model) zugewiesen. Nachfolgend beziehen Sie sich per Item... auf diesen Typ. Der Vorteil dieser Vorgehensweise ist neben der sichereren Verwendung auch die IntelliSense, die Ihnen mühsames Nachschlagen erspart.

41.18 Model Binding

Wem die bisherigen Datenbindungsvarianten noch nicht genug waren oder wer mehr Einfluss auf den Datenaustausch zwischen Control und Datenquelle haben will, der kann sich ab ASP.NET 4.5 auch einmal mit dem Thema "Model Binding" beschäftigen.

Vergessen Sie zunächst gänzlich alle bisherigen Ansätze, bis auf die *ObjectDataSource* (siehe Abschnitt 41.5, Seite 1995). Diese bot die Möglichkeit, ein Objekt mit geeigneten Schnitstellenmethoden (Select, Insert, Update, Delete) als Datenquelle zu nutzen.

Einen ähnlichen Ansatz bietet das Model Binding, mit dem Unterschied, dass Sie auf eine *ObjectDataSource* verzichten können und die Schnittstellenmethoden direkt in der Code-behind-Datei unterbringen. Die Microsoft-Entwickler versprechen sich dadurch einen höheren Einfluss auf das Datenhandling, es sollte aber auch nicht verschwiegen werden, dass dadurch eine unselige Vermischung von Code und Oberfläche stattfindet.

Ein Beispiel, basierend auf einem Entity-Data-Model soll für einen ersten Überblick genügen:

BEISPIEL 41.26: Model Binding

HTML

```
...
      <asp:GridView ID="GridView1" runat="server"
```

Zuweisen des Datentyps:

```
            ItemType="NorthwindModel.Products"
```

Auswahl der Daten:

```
            SelectMethod="GetProducts"
```

Aktualisieren der Daten:

```
            UpdateMethod="UpdateProduct"
```

Löschen der Daten:

```
            DeleteMethod="DeleteProduct"
            CellPadding="4" ForeColor="#333333" GridLines="None" PageSize="10"
```

Wir ermöglichen auch Paging und Sortieren:

```
            AllowPaging="True" AllowSorting="True"
...
       </asp:GridView>
...
```

HINWEIS: Beachten Sie, dass **keine** DataSource verwendet wird, alle Aufgaben werden durch die obigen Methoden umgesetzt!

C#

Die Select-Methode bzw. das Bereitstellen der Daten:

```
public IQueryable<NorthwindModel.Products> GetProducts()
{
    NorthwindModel.NorthwindEntities db = new NorthwindModel.NorthwindEntities();
    var prod = db.Products.OrderBy(p => p.ProductName);
    return prod;
}
```

BEISPIEL 41.26: Model Binding

Rückgabewert ist eine *IQueryable*-Collection. Dies ist von zentraler Bedeutung im weiteren Ablauf, da das Data-Control basierend darauf neue Abfragen bildet (z.B. für das Sortieren oder für die Realisierung des Paging). Es handelt sich also nicht um eine statische Liste, die hier zurückgegeben wird.

Beim Aktualisieren wird vom DataControl das gewünschte Element an die Methode übergeben, Ihre Aufgabe ist es, die Änderung in der Datenbasis zu realisieren:

```
public void UpdateProduct(NorthwindModel.Products newprod)
{
    NorthwindModel.NorthwindEntities db = new NorthwindModel.NorthwindEntities();
    NorthwindModel.Products oldprod = db.Products.Where(p =>
                    p.ProductID == newprod.ProductID).SingleOrDefault();
    oldprod.UnitsInStock = newprod.UnitsInStock;
    db.SaveChanges();
}
```

Im obigen Fall ändern wir nur die Produktanzahl, egal, was der Nutzer im Grid ändert.

Das Löschen sollte eigentlich auf dem gleichen Weg funktionieren, leider ist es den Autoren nicht gelungen, dies mit der vorliegenden Version zu realisieren. Der Übergabeparameter war immer null.

```
public void DeleteProduct(NorthwindModel.Products delprod)
{
    NorthwindModel.NorthwindEntities db = new NorthwindModel.NorthwindEntities();
    NorthwindModel.Products prod = db.Products.Where(p =>
              p.ProductID == delprod.ProductID).SingleOrDefault();
    db.DeleteObject(prod);
    db.SaveChanges();
}
```

Mehr zum Thema finden Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012].

41.19 Chart

Nachdem in der aktuellen .NET-Framework-Version bereits die Windows Forms-Programmierer[1] mit einer *Chart*-Komponente beglückt wurden, steht über den Namespace *System.Web.UI.DataVisualization.Charting* auch dem ambitionierten ASP.NET-Entwickler diese Möglichkeit offen.

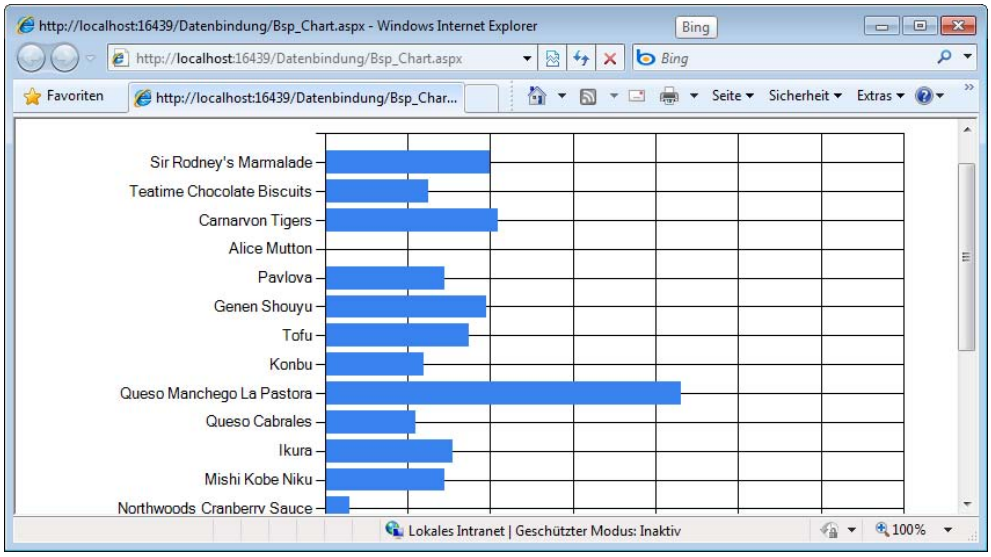
Ein kleiner Auszug aus der Liste der Möglichkeiten zeigt die Komplexität des Controls:

- 35 verschiedene Chart Typen
- eine unbegrenzte Anzahl von Chart-Bereichen, -Titeln, -Legenden etc.
- selbstverständlich die Möglichkeit der Datenbindung

[1] Auch die Silverlight-Programmierer!

- 3-D-Unterstützung
- diverse Skalierungsfunktionen
- mehr als 50 Funktionen aus dem Bereich Finanzmathematik und Statistik
- ...

Die folgende Abbildung zeigt ein *Chart*-Control in Aktion, die Anbindung an die *Products*-Tabelle der *Northwind*-Datenbank erfolgte per *SqlDataSource* (zusätzlich haben wir die Anzahl der Datensätze mit TOP etwas beschränkt).



Interessant ist sicherlich, in welcher Form die Daten auf dem Client angezeigt werden. Ein Blick in den Quellcode der Seite zeigt uns, dass es sich um ein PNG, d.h. eine auf dem Server generierte Grafik handelt, die als einfaches Image angezeigt wird.

BEISPIEL 41.27: Generierter HTML-Quellcode

HTML

```
...
<img id="Chart1" src="/Datenbindung/ChartImg.axd?
    i=chart_5de2a481b2e5451f8b3b98affca2bedc_0.png&amp;g=56d04bb07d8d4ccfb46446a97b04b824"
     alt="" style="height:600px;width:800px;border-width:0px;" />
...
```

HINWEIS: Wesentlich mehr zum Thema erfahren Sie in unserem Buch [Datenbankprogrammierung mit Visual C# 2012].

Kapitel 42

ASP.NET-Objekte/-Techniken

Das vorliegende Kapitel widmet sich den Themen

- Fehlerbehandlung/Debugging
- Mail-Versand
- Sicherheit von Webanwendungen
- AJAX
- Entwurf von User Controls

Dabei kommen zahlreiche für den ASP.NET-Programmierer wichtigen Objekte/Klassen zum Einsatz.

42.1 Wichtige ASP.NET-Objekte

Im Folgenden wollen wir Ihnen eine kurze Übersicht über die Verwendung bzw. Funktionsweise der wichtigsten ASP.NET-Objekte geben. Gemeint sind in diesem Fall nicht die sichtbaren Controls, mit denen Sie ja bereits Bekanntschaft geschlossen haben, sondern die im Hintergrund agierenden Klassen und Objekte, die unter anderem für die Ablaufsteuerung und den Datenaustausch zwischen Client und Server zuständig sind.

42.1.1 HTTPApplication

Mit dem *HttpApplication*-Objekt können Sie alle Objekte und Ereignisse, die die gesamte Anwendung betreffen, für alle Nutzer verwalten. Gemeint sind unter anderem folgende Klassen:

- *Application*
- *Request*
- *Response*
- *Server*

- *Session*
- *Site*
- *User*

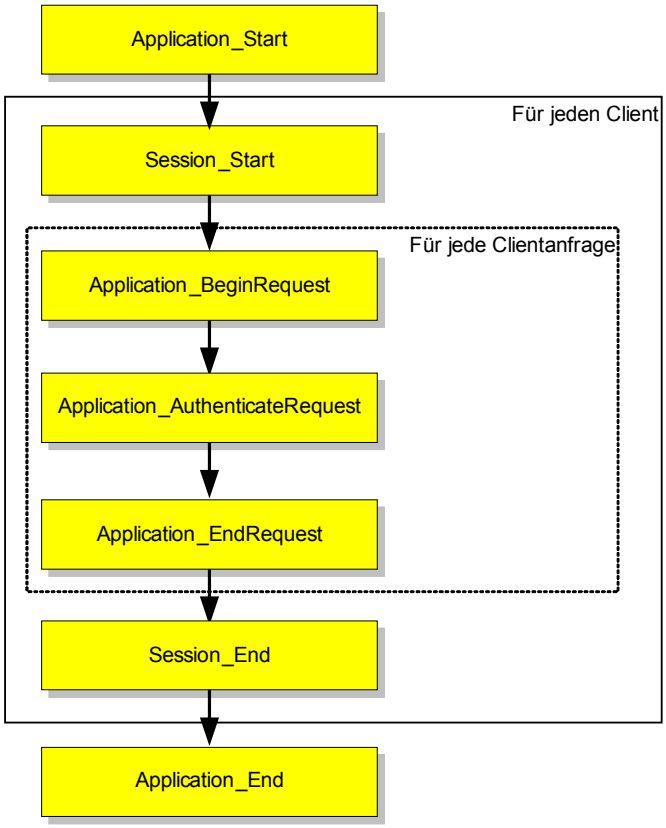
HINWEIS: Instanzen dieser Klasse werden direkt von ASP.NET und nicht durch Ihr Programm erstellt! Es wird jeweils nur eine Instanz der *HttpApplication*-Klasse für die Lebensdauer der Anwendung erzeugt.

Weiterhin bietet sich das *HTTPApplication*-Objekt mit seinen Ereignishandlern für die Initialisierung und Ablaufsteuerung Ihres Projekts an. Die zugehörigen Ereignis-Methoden sind in der Datei *Global.asax* definiert:

```
<%@ Application Language="C#" %>

<script runat="server">

    void Application_Start(object sender, EventArgs e)
    {
        // Code, der beim Starten der Anwendung ausgeführt wird.
    }

    void Application_End(object sender, EventArgs e)
    {
        //  Code, der beim Beenden der Anwendung ausgeführt wird.
    }

    void Application_Error(object sender, EventArgs e)
    {
        // Code, der bei einem nicht behandelten Fehler ausgeführt wird.
    }

    void Session_Start(object sender, EventArgs e)
    {
        // Code, der beim Starten einer neuen Sitzung ausgeführt wird.
    }

    void Session_End(object sender, EventArgs e)
    {
        // Code, der am Ende einer Sitzung ausgeführt wird.
    }

</script>
```

HINWEIS: Diese Datei gehört nicht standardmäßig zum Webprojekt, Sie müssen sie zunächst über den Menüpunkt *Website|Neues Element hinzufügen|Globale Anwendungsklasse* erzeugen.

Die Reihenfolge der Abarbeitung obiger Methoden:



Optional stehen Ihnen noch folgende Ereignisse zur Verfügung:

- *BeginRequest*
- *AuthenticateRequest*
- *AuthorizeRequest*
- *ResolveRequestCache*
- *AcquireRequestState*
- *PreRequestHandlerExecute*
- *PostRequestHandlerExecute*
- *ReleaseRequestState*
- *UpdateRequestCache*
- *EndRequest*

HINWEIS: Die entsprechenden Ereignishandler müssen Sie selbst erzeugen, dafür sind keine Vorlagen vorhanden!

42.1.2 Application

Mit Hilfe dieses Objekts (es ist bereits instanziiert) können Sie innerhalb Ihrer Anwendung Daten temporär zwischenspeichern. Im Unterschied zur *Session* (siehe folgender Abschnitt) stehen diese Daten allen Nutzern der Webanwendung zur Verfügung.

Auf die Collection können Sie per Index oder über einen Namen (quasi der Variablenname) zugreifen.

BEISPIEL 42.1: Zugriff auf *Application*-Variablen

C#
```
Application["Name"] = "Ich bin es ...";
Application.Add("Vorname", "Hier noch der Vorname");
TextBox1.Text = (string) Application["Name"];
TextBox2.Text = (string) Application[2];              // Zugriff auf Vorname
```

HINWEIS: Wie Sie den beiden letzten Zeilen entnehmen, muss beim Auslesen eine Typisierung erfolgen, denn es werden nur Objekte abgespeichert.

BEISPIEL 42.2: Die Startzeit der Applikation wird abgespeichert und in einem späteren Formularaufruf angezeigt.

C#

In der *Global.asax*:

```
<%@ Application Language="C#" %>

<script runat="server">
    void Application_Start(object sender, EventArgs e)
    {
        Application.Add("Startzeit", System.DateTime.Now);
    }
...
```

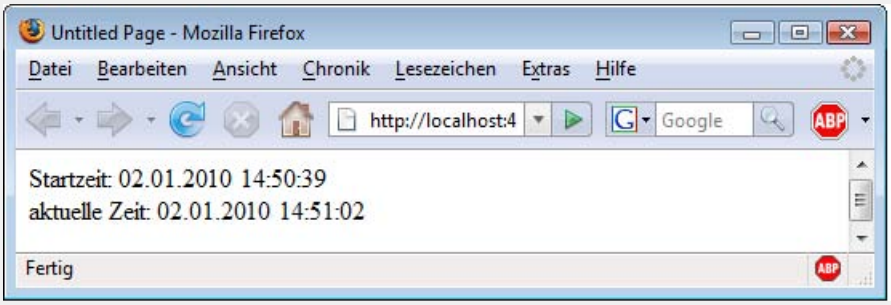
Im Formular:

```
...
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write("Startzeit: " + Application["Startzeit"]+ "<br>");
        Response.Write("aktuelle Zeit: " + System.DateTime.Now.ToString() + "<br>");
    }
...
```

BEISPIEL 42.2: Die Startzeit der Applikation wird abgespeichert und in einem späteren Formularaufruf angezeigt.

Ergebnis

Ein Test:



HINWEIS: Verändern Sie einmal die Datei *Default.aspx* und starten dann das Projekt erneut. Die Startzeit dürfte gleich bleiben, die *Global.asax* wurde ja nicht verändert (keine neue Assembly, kein neuer Anwendungsstart). Eine kleine Änderung an der *Global.asax* reicht, um den Compiler in Aktion treten zu lassen und jetzt ändert sich auch die Startzeit wieder.

42.1.3 Session

Mit *Session* bezeichnet man die Sitzung eines Web-Users. Die Session wird initialisiert, wenn ein Nutzer die erste Seite Ihrer Applikation aufruft bzw. sich mit Ihrem Server verbindet, und endet nach dem Abruf der letzten Seite. Da dies ein relativer Zustand ist (wann ist es die letzte Seite?), lässt der Server einfach eine bestimmte Zeit verstreichen, bis die Session als beendet gilt.

HINWEIS: Sessions erfordern eine Session-ID auf dem Client!

Session-Variablen

Über einen eindeutigen Bezeichner verwaltet das *Session*-Objekt eine Liste von Objekten, und das können auch einfache Variablenwerte sein. Diese Objekte sind von allen Web-Forms Ihrer Anwendung aus erreichbar, also der ideale Weg, um Daten zwischen einzelnen Formularen auszutauschen.

BEISPIEL 42.3: Erzeugen einer *Session*-Variablen

C#

```
Session["Begrüßung"]  = "Hallo User!";
Session["Kontostand"] = 10000;
```

Oder auch so:

```
Session.Add("Ausgaben", 3456);
```

Im weiteren Verlauf können Sie diese Variablen auslesen, verändern oder auch löschen.

BEISPIEL 42.4: Auflisten aller Variablen

C#

```
protected void Page_Load(object sender, EventArgs e)
{
    Session["Begrüßung"]  = "Hallo User!";
    Session["Kontostand"] = 10000;
    Session.Add("Ausgaben", 3456);

    for (int i = 0; i < Session.Count; i++)
        Response.Write("Session-Variable (" + i.ToString() + ") = " +
                    Session[i].ToString() + "<br>");
}
```

Ergebnis

Das Ergebnis im Browser:

Session-Variable (0) = Hallo User!
Session-Variable (1) = 10000
Session-Variable (2) = 3456

Beenden einer Session

Drei Wege bieten sich an:

- Sie warten einfach die vom IIS vordefinierte Zeit ab, bis für den Server die Session als beendet angesehen wird.
- Sie definieren über die Datei *Web.config* einen neuen *Timeout*-Wert.
- Sie verwenden die *Session.Timeout*-Eigenschaft, um einen neuen *Timeout*-Wert festzulegen (in Minuten).
- Sie beenden die Session direkt über den Aufruf der Methode *Session.Abandon*.

Ereignisse

Über die Datei *Global.asax* stehen Ihnen die folgenden Ereignisse im Zusammenhang mit der Session zur Verfügung:

```
void Session_Start(object sender, EventArgs e)
{
    // Code, der beim Starten einer neuen Sitzung ausgeführt wird.
}

void Session_End(object sender, EventArgs e)
{
    // Code, der am Ende einer Sitzung ausgeführt wird.
}
```

42.1.4 Page

Hierbei handelt es sich um die bereits mehrfach verwendete Web-Form-Seite, die Sie zwar im Designer entwerfen (*aspx*-Datei), die jedoch erst zur Laufzeit vom Server erzeugt und an den Client (Browser) gesendet wird. Auf dem Server wird dazu passend ein *Page*-Objekt erzeugt.

Page verfügt unter anderem über einige recht interessante Eigenschaften.

Eigenschaften

Eigenschaft	Beschreibung
Controls	… eine Auflistung aller enthaltenen Controls.
EnableViewState	… bestimmt, ob die Controls nach einem erneuten Seitenaufruf ihren Status und Inhalt behalten.
ErrorPage	… eine alternative Seite, die bei nicht behandelten Fehlern automatisch aufgerufen wird.
IsPostBack	… zeigt, ob es sich um den ersten oder einen erneuten Aufruf der Seite handelt (siehe Beispiel).
IsValid	… fasst das Ergebnis aller Validierungsversuche zusammen.
Validators	… eine Auflistung aller Validation-Controls der Seite.

BEISPIEL 42.5: Verwendung von *IsPostBack*, um die Anzahl der Seitenaufrufe durch den Nutzer zu ermitteln.

C#

Realisierung: Platzieren Sie lediglich einen Button auf dem Web-Form und fügen Sie dem Formular folgende Ereignisprozedur hinzu:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Page.IsPostBack)
    {
       Session["Count"] = ((int)Session["Count"])+ 1;
       Response.Write("Hallo, ich werde zum " + ((int) Session["Count"]).ToString()
                + ". Mal aufgerufen!");
    } else
    {
```

Die Seite wird zum ersten Mal aufgerufen:

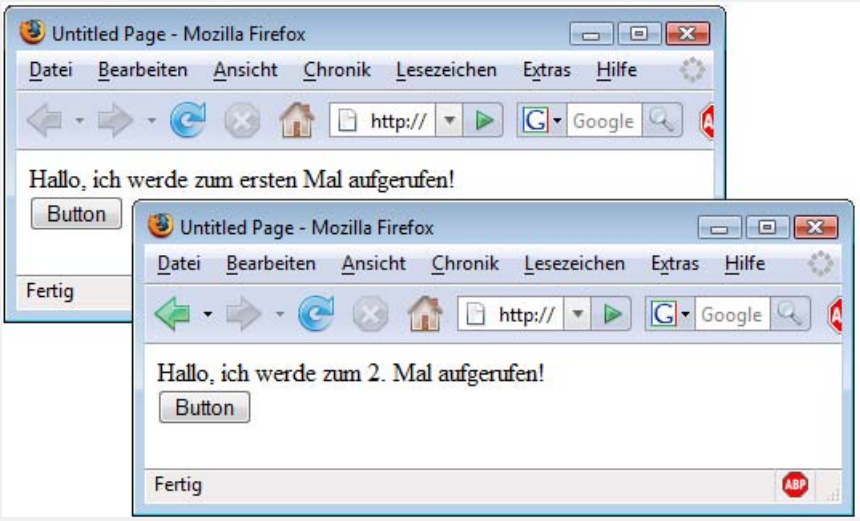
```
       Session["Count"] = 1;
       Response.Write("Hallo, ich werde zum ersten Mal aufgerufen!");
    }
 }
```

Wie Sie sehen, verwenden wir eine *Session*-Variable, um über den Seitenaufruf hinweg die Anzahl der Aufrufe zu zählen.

BEISPIEL 42.5: Verwendung von *IsPostBack*, um die Anzahl der Seitenaufrufe durch den Nutzer zu ermitteln.

Ergebnis

Starten Sie das Beispiel, dürfte Ihnen beim ersten Mal die Aufschrift "Hallo, ich werde zum ersten Mal aufgerufen!" angezeigt werden. Nach dem Klick auf den Button wird ein Aufruf an den Server gesendet, die Seite wird aktualisiert und die Beschriftung ändert sich entsprechend.



HINWEIS: Nutzen Sie diese Möglichkeit, wenn Sie lediglich beim ersten Aufruf der Seite eine Prozedur (Datenabruf aus der Datenbank) ausführen möchten!

Methoden

Einige wichtige Methoden der Klasse *Page*:

Methode	Beschreibung
DataBind	... realisiert die Datenbindung der Seite
FindControl	... sucht das angegebene Server Control
HasControls	... verfügt die Seite über Controls (Text oder Server Controls)?
LoadControll	... lädt ein User Control nach
MapPath	... wandelt virtuellen (URL) in absoluten Pfad um
ResolveUrl	... wandelt absoluten in virtuellen (URL) Pfad um
Validate	... löst die Validierung aller vorhandenen Validation Controls aus

BEISPIEL 42.6: Datenbindung

C#

Fügen Sie ein *Label*-Control (aus Standard) in *Default.aspx* ein. Wechseln Sie nachfolgend in die HTML-Quellcodeansicht und suchen Sie die Deklaration des Labels. Erweitern Sie diese um die folgenden Angaben:

```
<asp:Label ID="Label1" runat="server" Text="<%# myText %>">
```

In *Default.aspx.cs* ergänzen Sie die Klassendeklaration um folgenden Ausdruck:

```
...
public partial class _Default : System.Web.UI.Page
{
    public string myText = "Hallo";
```

In *Page_Load* können Sie jetzt die Bindung herstellen:

```
    protected void Page_Load(object sender, EventArgs e)
    {
        this.DataBind();
    }
```

Als Resultat wird in der finalen HTML-Seite der Text von *Label1* in "Hallo" geändert.

Ereignisse

Neben dem bereits mehrfach verwendeten *Load*-Ereignis können Sie noch auf die folgenden Events zugreifen:

Ereignis	Beschreibung
Disposed	… die Seite wird aus dem Speicher entfernt
Error	… ein nicht behandelter Fehler ist aufgetreten
Init	… die Seite wird initialisiert (vor *Load*)
Load	… die Seite wird geladen
PreRender	… die Seite soll gezeichnet werden
Unload	… die Seite wird entladen

Die o.g. Ereignis-Prozeduren können Sie recht einfach selbst deklarieren: Schreiben Sie einfach *Page_* vor den Ereignisnamen und ASP.NET stellt automatisch die Bindung zum gewünschten Ereignis her.

BEISPIEL 42.7: *PreRender*-Event

C#

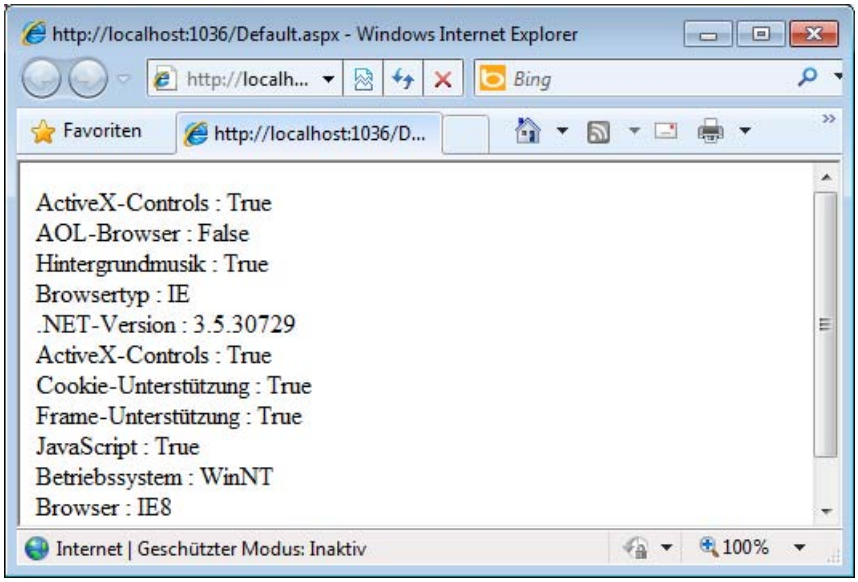
```
    void Page_PreRender()
    {
        Session["abc"] = "Hallo";
        Response.Write("Hallo Welt");
    }
```

42.1.5 Request

Über das bereits initialisierte *Request*-Objekt haben Sie aus dem Web-Form heraus Zugriff auf alle vom Client gesendeten Daten. So können Sie Informationen über die Anforderung selbst gewinnen oder auch über den Browser des Nutzers bzw. den Nutzer selbst (IP-Adresse).

Informationen über den Browser ermitteln

Verwenden Sie die *Browser*-Eigenschaft, um die Fähigkeiten des Clients zu ermitteln:



Der Quelltext:

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("ActiveX-Controls : " + Request.Browser.ActiveXControls + "<br>");
    Response.Write("AOL-Browser : " + Request.Browser.AOL + "<br>");
    Response.Write("Hintergrundmusik : " + Request.Browser.BackgroundSounds + "<br>");
    Response.Write("Browsertyp : " + Request.Browser.Browser + "<br>");
    Response.Write(".NET-Version : " + Request.Browser.ClrVersion.ToString() + "<br>");
    Response.Write("ActiveX-Controls : " + Request.Browser.ActiveXControls + "<br>");
    Response.Write("Cookie-Unterstützung : " + Request.Browser.Cookies + "<br>");
    Response.Write("Frame-Unterstützung : " + Request.Browser.Frames + "<br>");
    Response.Write("JavaScript : " + Request.Browser.JavaScript + "<br>");
    Response.Write("Betriebssystem : " + Request.Browser.Platform + "<br>");
    Response.Write("Browser : " + Request.Browser.Type + "<br>");
}
```

Weitere Informationen lassen sich über die folgenden Eigenschaften des Objekts sammeln:

Eigenschaft	Beschreibung
ApplicationPath	… virtueller Server-Pfad der Webanwendung
Browser	… siehe oben
ContentLength	… Bytes, die vom Client gesendet wurden
ContentType	… MIME-Type des Requests
Cookies	… vom Client gesendete Cookies
FilePath	… gibt den virtuellen Pfad der aktuellen Client-Anfrage zurück
Form	… bietet Zugriff auf Form-Inhalte einer POST-Anfrage
InputStream	… Stream mit den Anfragedaten
IsAuthenticated	… erfolgreich eingeloggt?
IsSecureConnection	… handelt es sich um eine HTTPS-Verbindung?
QueryString	… Abfrage der per URL-Parameter übergebenen Variablen
RequestType	… Typ der Anfrage (GET oder POST)
ServerVariables	… Zugriff auf Servervariablen
TotalBytes	… Bytes im *InputStream*
Url	… Informationen über den URL der aktuellen Anfrage
UserHostAddress	… die IP-Adresse des Clients
UserHostName	… der Hostname des Clients
UserLanguages	… vom Client bevorzugte Sprachen

BEISPIEL 42.8: Client-Adresse und -Sprache(n) abfragen

C#

```
private void Page_Load(object sender, System.EventArgs e)
{
   Response.Write("Adresse : " + Request.UserHostAddress + "<br>");
   for (int i = 0; i < Request.UserLanguages.Length; i++)
     Response.Write("Sprachen : " + Request.UserLanguages[i] + "<br>");
}
```

Ergebnis

Die Ausgabe:

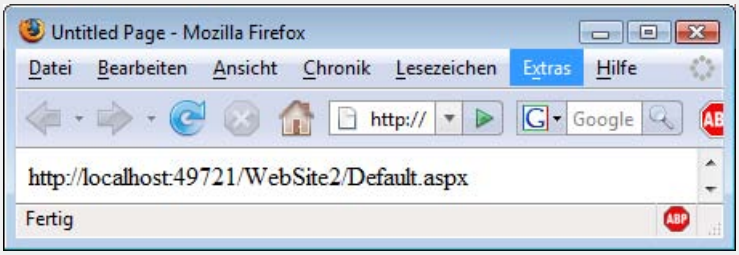
```
Adresse : 127.0.0.1
Sprachen : de-de
Sprachen : de;q=0.8
Sprachen : en-us;q=0.5
Sprachen : en;q=0.3
```

BEISPIEL 42.9: Die Adresse der aktuellen Anwendung/Seite bestimmen

C#

```
    protected void Page_Load(object sender, EventArgs e)
    {
       Response.Write(Request.Url.ToString());
...
```

Ergebnis



QueryString-Parameter auslesen

Haben Sie bereits mit ASP gearbeitet, werden Ihnen die *QueryString*-Parameter noch in Erinnerung sein. Hierbei handelt es sich im Prinzip um Variablen, die per URL von einer Webseite an die nächste übergeben werden können. Realisiert wird dies durch ein an den URL angehängtes Fragezeichen, darauf folgt der Variablenname, ein Gleichheitszeichen sowie der Wert. Weitere Variablen lassen sich nach einem "&"-Zeichen nach obigem Schema anhängen.

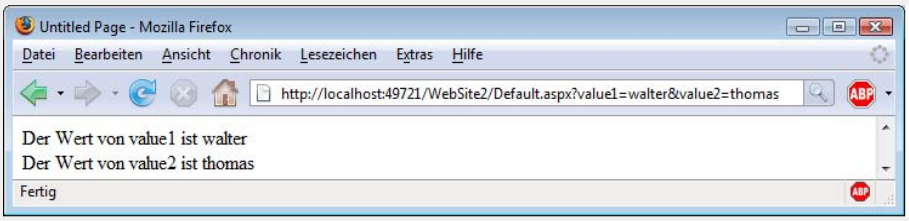
BEISPIEL 42.10: Abfrage aller Parameter und Parameterwerte

C#

```
    protected void Page_Load(object sender, EventArgs e)
    {
       for (int i = 0; i < Request.QueryString.Count; i++)
          Response.Write("Der Wert von " + Request.QueryString.Keys[i] + " ist " +
                   Request.QueryString[Request.QueryString.Keys[i]] + "<br>");
    }
```

Ergebnis

Rufen Sie die Seite mit einem entsprechend erweiterten URL auf, sollten Sie ein Ergebnis wie in der folgenden Abbildung erhalten:



42.1.6 Response

Mit dem *Response*-Objekt (Antwort, Rückmeldung) haben Sie den Gegenpart des *Request*-Objekts vor sich. Alle vom Server an den Client zu sendenden Informationen laufen über dieses Objekt. Damit kommt dem *Response*-Objekt eine zentrale Stellung beim Datenaustausch zu, auch wenn die Bedeutung mit der Einführung von ADO.NET etwas zurückgegangen ist. Neben dem reinen Datentransfer bietet das Objekt auch die Möglichkeit, den Client mit den mehr oder weniger beliebten Cookies zu beglücken.

Eigenschaften

Die wichtigsten Eigenschaften auf einen Blick:

Eigenschaft	Beschreibung
BufferOutput	… soll die Seite komplett (*True*) oder häppchenweise (*False*) an den Client gesendet werden? Ist der Wert auf *True* gesetzt, werden alle Anweisungen zwischengespeichert und erst am Ende zum Client gesendet.
Charset	… der Zeichensatz für den Antwortstream.
ContentEncoding	… legt die Kodierung der Antwortseite fest.
ContentType	… legt den MIME-Type der Antwort fest (z.B. *image/jpeg*).
Cookies	… alles rund um die beliebten Cookies.
ExpiresAbsolute	… Datum für den Verfall der auf dem Client gecachten Webseite.
IsClientConnected	… ist der Client noch "am Leben"?
Output	… Senden von Textdaten an den Client.
OutputStream	… ein *Stream*-Objekt für die Ausgabe binärer Daten (z.B. Grafiken).

Methoden

Natürlich finden Sie auch reichlich Methoden (Auszug):

Methode	Beschreibung
AppendHeader	… fügt der Antwortkopfzeile einen neuen Eintrag hinzu
Clear	… löscht den gesamten Antwortpuffer
ClearContent	… löscht den Antwortpuffer, Kopfzeilen bleiben erhalten
ClearHeaders	… löscht die Kopfzeilendaten aus dem Antwortpuffer
Close	… schließt die Client-Server-Verbindung
Flush	… sendet Antwortpufferdaten an den Client
Redirect	… der Client wird aufgefordert, den übergebenen URL abzurufen
Write	… sendet eine Zeichenkette an den Client
WriteFile	… die angegebene Datei wird an den Client gesendet

Textausgabe

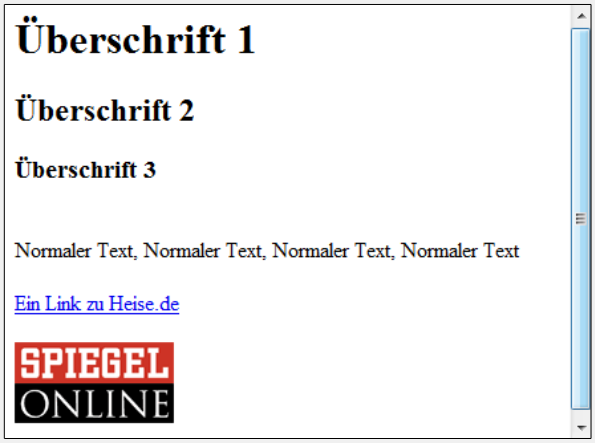
In den vorhergehenden Abschnitten wurde ja schon rege von der *Write*-Methode Gebrauch gemacht. Neben einfachen Textausgaben können Sie damit auch gezielt HTML-Tags etc. an den Client senden, so als ob Sie den HTML-Seitentext mit einem Editor bearbeiten würden.

BEISPIEL 42.11: Verschiedene Ausgaben an den Client senden

C#

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("<H1>Überschrift 1</H1>");
    Response.Write("<H2>Überschrift 2</H2>");
    Response.Write("<H3>Überschrift 3</H3><br>");
    Response.Write("Normaler Text, Normaler Text, Normaler Text, Normaler Text<br><br>");
    Response.Write("<A HREF=\"http://www.heise.de\">Ein Link zu Heise.de</A><br><br>");
    Response.Write("<img src=\"http://www.spiegel.de/static/sys/logo_120x61.gif\">");
}
```

Ergebnis



Bilder/Dateien zum Client senden

Im Gegensatz zur *Write*-Methode können Sie mit *WriteFile* eine komplette Datei an den Client senden. Dazu müssen Sie lediglich den Dateinamen übergeben. Vorher sollten Sie mit *ContentType* noch den Datentyp festlegen.

BEISPIEL 42.12: Eine Grafik an den Client senden (diese wird automatisch angezeigt)

C#

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.ContentType = "image/gif";
    Response.WriteFile(@"C:\Inetpub\wwwroot\objekte\test.gif");
}
```

Dynamisch erzeugte Grafiken senden

Nicht in jedem Fall liegt die Grafik oder Datei, die Sie an den Client senden wollen, bereits vor. Möchten Sie beispielsweise einen optisch "aufgemotzten" Counter oder Ähnliches realisieren, müssen Sie einen anderen Weg beschreiten[1].

Die Lösung ist mit der *Stream*-Eigenschaft des *Response*-Objekts schnell gefunden. Da auch die diversen Grafikobjekte *Streams* unterstützen, steht einer professionellen Realisierung nichts im Wege.

BEISPIEL 42.13: Ein grafischer "Counter"

C#

```
...
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
...
private void Page_Load(object sender, System.EventArgs e) {
```

Bitmap erzeugen:

```
   Bitmap img = new Bitmap(200, 100);
```

Graphics-Objekt für Bitmap erzeugen:

```
   Graphics g = Graphics.FromImage(img);
```

Einen *GradientBrush* erzeugen:

```
   Rectangle rect = new Rectangle(0, 0, 200, 150);
   LinearGradientBrush myBrush  = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);
```

Bitmap-Hintergrund zeichnen:

```
   g.FillRectangle(myBrush, new Rectangle(0, 0, 200, 100));
```

Den gewünschten Text ausgeben:

```
   myBrush = new LinearGradientBrush(rect, Color.Yellow, Color.Red,
                LinearGradientMode.Vertical);
   g.DrawString("11999", new Font("Arial", 48), myBrush, 2, 2);
```

Datentyp festlegen und Daten versenden:

```
   Response.ContentType = "image/jpeg";
   img.Save(Response.OutputStream, ImageFormat.Jpeg);
}
```

[1] Sehen wir einmal von der "Billiglösung" einer temporären Datei ab.

BEISPIEL 42.13: Ein grafischer "Counter"

Ergebnis

Neue Seiten aufrufen

Vermutlich haben Sie schon vergeblich nach einem *Show* oder *ShowModal* Ausschau gehalten, wenn Sie weitere Web Formulare anzeigen wollten.

Neben dem trivialen Einbinden eines Hyperlinks in das Startformular bietet sich auch ein Redirect an. Basierend auf dem HTTP-Protokoll wird an den Client der Statuscode 302 "Object moved" gesendet, um **den Client** zu veranlassen, eine in der Kopfzeile der Meldung mitgelieferte Webadresse aufzurufen. Die Weiterleitung hat also einen erneuten Aufruf des Browsers zur Folge, mit dem Unterschied, dass dieser jetzt die gewünschte Seite aufruft.

BEISPIEL 42.14: Nach einem Klick auf einen Button soll *WebForm2.aspx* aufgerufen werden

C#

```
protected void Button1_Click(object sender, EventArgs e)
{
   Response.Redirect("Seite2.aspx");
}
```

HINWEIS: Alternativ können Sie auch serverseitige Lösungen, wie *Server.Transfer* oder *Server.Execute*, verwenden, um eine andere Webseite aufzurufen bzw. an den Client zu senden.

```
protected void Button1_Click(object sender, EventArgs e)
{
   Server.Transfer("Seite2.aspx");
}
```

HINWEIS: Werfen Sie nach dem obigen Klick-Ereignis mal einen Blick auf die Adresszeile des Browsers. Sie erhalten zwar die gewünschte *Seite2*, aber in der Adresszeile steht nach wie vor der Name der aufrufenden Seite.

Wir geben zur Protokoll ...

Geht es um das Speichern von Protokolldaten, bietet sich die Methode *AppendToLog* an. Übergeben Sie einfach die gewünschte Zeichenkette, um diese im Log-File des IIS abzuspeichern.

BEISPIEL 42.15: Speichern der Useradresse, wenn auf den Button geklickt wird.

C#

```
protected void Button1_Click(object sender, EventArgs e)
{
    Response.Write("Ausgaben der Seite 2: <br><br>");
    Response.AppendToLog("Zugriff von " + Request.UserHostAddress.ToString());
}
```

Ergebnis

Der Protokollinhalt:

```
127.0.0.1, -, 01.01.2005, 20:33:35, W3SVC1, P4, 127.0.0.1, 0, 301, 830, 200, 0, GET,
/Diverses/WebForm1.aspx, -,
127.0.0.1, -, 01.01.2005, 20:33:35, W3SVC1, P4, 127.0.0.1, 0, 703, 1007, 200, 0, POST,
/Diverses/WebForm1.aspx, Zugriff+von+127.0.0.1,
127.0.0.1, -, 01.01.2005, 20:33:37, W3SVC1, P4, 127.0.0.1, 0, 703, 1007, 200, 0, POST,
/Diverses/WebForm1.aspx, Zugriff+von+127.0.0.1,
...
```

42.1.7 Server

Das Server-*Objekt* (Klasse *HTTPServerUtility*) bietet Ihnen ein Sammelsurium von kleinen Hilfsroutinen, die in der täglichen Arbeit häufig gebraucht werden. Einige kleine Beispiele zeigen die Verwendung.

MapPath

BEISPIEL 42.16: Den absoluten Datenbankpfad auf dem Server bestimmen

C#

```
Response.Write(Server.MapPath("./App_Data") + "<br>");
```

Ergebnis

Die Ausgabe im Browser:

C:\temp\WebSite2\App_Data

Sie können auch gleich den kompletten Pfad einer Datei bestimmen:

```
Response.Write(Server.MapPath("./App_Data/Daten.xml") + "<br>");
```

MachineName

BEISPIEL 42.17: Bestimmen des Servernamens

C#
```
Response.Write(Server.MachineName + "<br>");
```

Transfer

BEISPIEL 42.18: Weiterleiten einer Clientanfrage an eine andere Adresse

C#
```
Server.Transfer("Seite2.aspx");
```

HINWEIS: In der Adresszeile steht weiterhin die Adresse der aufrufenden Seite, der Client bekommt von den "Manipulationen" auf dem Server nichts mit!

Execute

BEISPIEL 42.19: Einfügen einer Seite

C#
```
Response.Write("Hier ist Seite1 Zeile1<br>");
Server.Execute("Seite2.aspx");
Response.Write("Hier ist Seite1 Zeile3<br>");
```

Ergebnis

Hier ist Seite1 Zeile1
Hier ist Seite2 Zeile1

Hier ist Seite1 Zeile3

HINWEIS: Wie Sie sehen, wird die Bearbeitung der *Seite1* nach dem *Execute* fortgesetzt.

42.1.8 Cookies verwenden

Möchten Sie die Nutzer Ihrer Webseiten mit Cookies peinigen, können Sie mit der Klasse *HttpCookie* problemlos[1] neue "Kekse" produzieren.

Eigenschaft	Beschreibung
Domain	Der Domain-Name von dem der Cookie erstellt wird.
Expires	Das Verfallsdatum des Cookies. Geben Sie keinen Wert an, ist der Cookie temporär und wird beim Beenden der Browsersitzung gelöscht.
HasKeys	Besitzt der Cookie Unterschlüssel?
Name	Ein eindeutiger Name für den Cookie, auch Parameter im Konstruktor.

[1] Wenn der genervte User dies zulässt.

Eigenschaft	Beschreibung
Path	Der Pfad des Cookies innerhalb der Domain.
Secure	(Default *False*) Cookies werden nur über sichere Verbindungen zurückgesendet.
Value	Der eigentliche Wert, der im Cookie zu speichern ist.
Values	Weitere Werte, die im Cookie als *NameValueCollection* gespeichert werden.

BEISPIEL 42.20: Erzeugen eines Cookies mit der aktuellen Anmeldezeit am Server

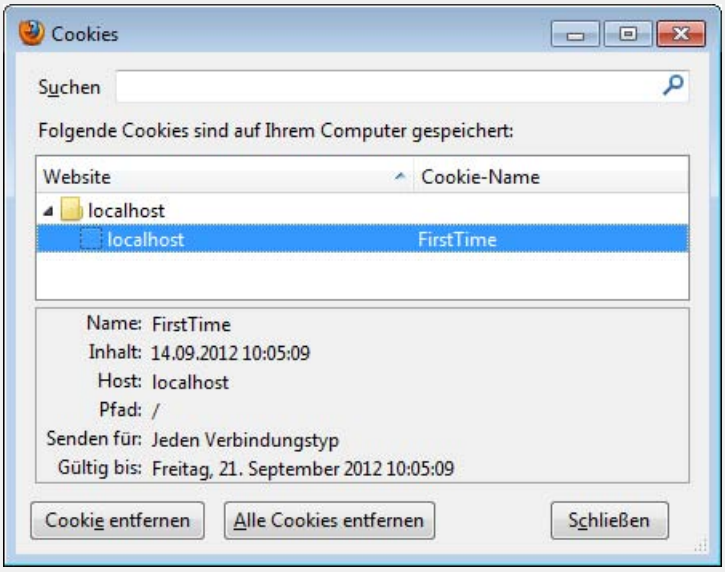
C#

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!Page.IsPostBack)
        {
            HttpCookie Cookie = new HttpCookie("FirstTime");
            Cookie.Value = DateTime.Now.ToString();
            Cookie.Expires = DateTime.Now.AddDays(7);
            Response.Cookies.Add(Cookie);
            Response.Write("Willkommen");
        }
}
```

In der *Value*-Eigenschaft speichern wir den aktuellen Zeitstempel ab, der Cookie selbst ist für die nächsten sieben Tage gültig.

Ergebnis

Nach dem Aufruf zeigt der Cache unseres Browsers folgenden Inhalt:



Das Ablaufdatum ist korrekt gesetzt, der Inhalt wird oben angezeigt.

Doch wie können wir beim nächsten Mal auf diesen Cookie zugreifen? Diesmal ist nicht das *Response*-, sondern das *Request*-Objekt gefragt, da wir die Information nicht senden, sondern vom Client einlesen wollen.

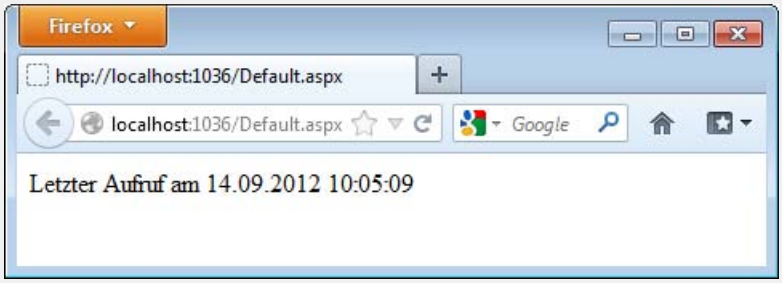
BEISPIEL 42.21: Lesen des Cookies

C#

```
private void Page_Load(object sender, System.EventArgs e)
{
   if (!Page.IsPostBack)
        Response.Write("Letzter Aufruf am " + Request.Cookies["FirstTime"].Value);
}
```

Ergebnis

Die Ausgabe im Browser:



Doch was ist, wenn der Cookie nicht existiert? Die einfachste Lösung bietet sich mit einer Fehlerbehandlung an:

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!Page.IsPostBack) try
        {
           Response.Write("Letzter Aufruf am " + Request.Cookies["FirstTime"].Value);
        }
        catch
        {
            HttpCookie Cookie = new HttpCookie("FirstTime");
            Cookie.Value = DateTime.Now.ToString();
            Cookie.Expires = DateTime.Now.AddDays(7);
            Response.Cookies.Add(Cookie);
            Response.Write("Willkommen");
        }
}
```

Löschen Sie ruhig einmal den Cookie und probieren Sie, was passiert[1].

[1] Wenn Sie viel Zeit haben, können Sie natürlich auch sieben Tage warten, dann ist der Cookie verfallen.

User

An dieser Stelle müssen wir Sie etwas vertrösten, im Abschnitt 42.4 gehen wir detailliert und im Zusammenhang auf das *User*-Objekt ein.

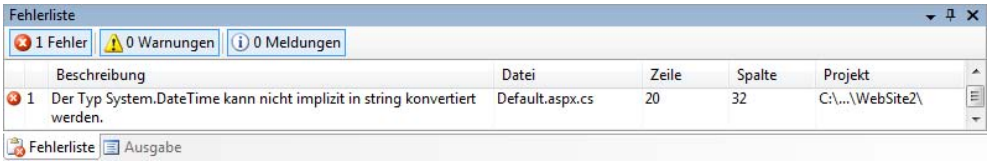
42.2 Fehlerbehandlung unter ASP.NET

Warum sollten ausgerechnet in ASP.NET-Anwendungen keine Fehler auftreten? Das Gegenteil ist der Fall, und Sie werden bestimmt schon die Bekanntschaft der Fehlerseiten gemacht haben, wenn Sie einige der vorhergehenden Beispiele nachvollzogen haben.

42.2.1 Fehler beim Entwurf

Hier wird es in ASP.NET im Gegensatz zu ASP ganz einfach. Wie bei einer "normalen" Windows-Anwendung "meckert" der Compiler, wenn Sie neue Eigenschaften oder Objekte "erfunden" oder Parameter vergessen haben. In ASP-Anwendungen durften Sie solche Überraschungen erst nach einem intensiven Test erleben.

Die Fehlermeldung eines ASP.NET-Projekts verdeutlicht die Vorgehensweise:



Nach einem simplen Doppelklick auf die Meldung sind Sie auch schon an der richtigen Stelle im Quellcode gelandet:

```
HttpCookie Cookie = new HttpCookie("FirstTime");
Cookie.Value = DateTime.Now;
Cookie.Expires
Response.Cookies.Add(Cookie);
Response.Write("Willkommen");
```

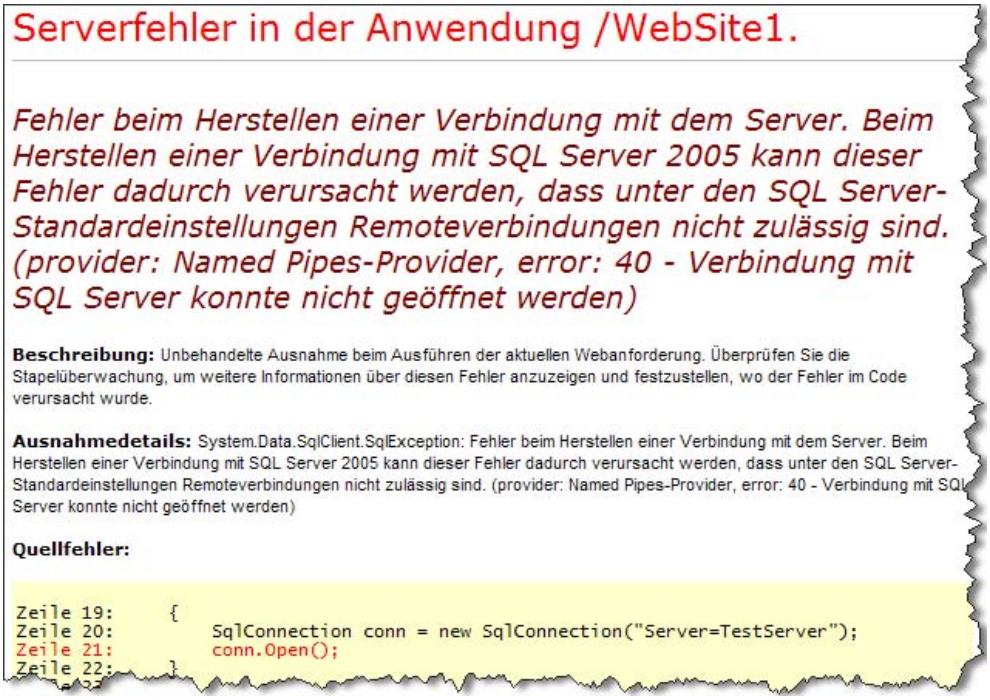
Der Typ System.DateTime kann nicht implizit in string konvertiert werden.

Einfacher geht es wohl nicht mehr, eine automatische Fehlerkorrektur möchten Sie bestimmt nicht haben.

42.2.2 Laufzeitfehler

Doch was passiert eigentlich, wenn es sich um einen der "gemeinen" Laufzeitfehler handelt? Sei es, dass eine Datei auf dem Server nicht gefunden wurde, ein Passwort nicht stimmt oder gerade der Datenbank-Server offline ist. Die Antwort zeigt Ihnen ein kurzer Test.

Das Ergebnis einer fehlgeschlagenen Verbindungsaufnahme zu einem SQL-Server (Server existiert nicht) zeigt die folgende Abbildung.



Der Browser wird geradezu mit Informationen über die Fehlerursache voll gestopft. Neben der ausführlichen Fehlermeldung wird auch gleich der zugehörige Quellcode angezeigt.

Doch stopp! So viele Informationen wollen Sie dem Endanwender Ihrer Web-Applikation sicher nicht zukommen lassen. Fehlte noch, dass eventuell Passwörter etc. angezeigt werden. Doch keine Angst, eine derart ausführliche Fehlermeldung erhalten Sie nur, wenn Sie die Anwendung von Ihrem Entwicklungsrechner aus starten. Nutzen Sie einen anderen PC, wird Ihnen eine ganz andere Meldung angezeigt, wie die folgende Abbildung veranschaulicht.

Die Aussage "*ASP verbirgt den Quellcode vor dem Anwender*" stimmt also nach wie vor. Doch auch die obige Seite dürfte beim Nutzer keinen guten Eindruck hinterlassen, die Webseite erinnert fatal an gewisse mysteriöse Dialogboxen, die manchen Windows-Anwender in Angst und Schrecken versetzt haben.

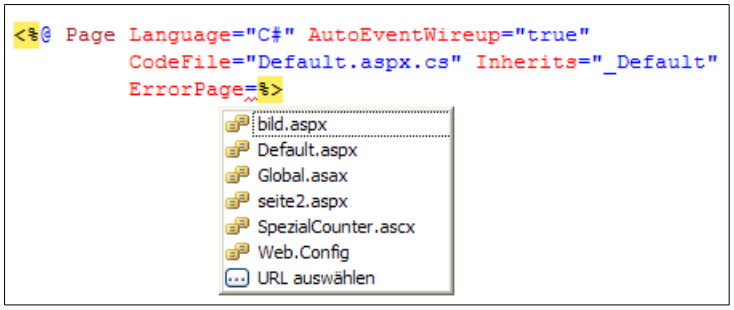
HINWEIS: Wie bereits erwähnt, können Sie die Fehlerausgabe mit Hilfe der *Web.config*-Datei auch so konfigurieren, dass die ausführlichen Fehlermeldungen ebenfalls auf allen anderen PCs angezeigt werden. Setzen Sie dazu im <*customerrors*>-Tag das Attribut *mode* auf "Off".

Besser und vor allem professioneller ist eine eigene Fehlermeldung, doch die gute alte *MessageBox* dürfte Ihnen da kaum weiterhelfen – Sie erinnern sich, dass der Code auf dem Server ausgeführt wird, und dort könnte mit der Meldung sowieso keiner etwas anfangen, es sei denn, Sie schreiben die Informationen in das System-Log.

42.2.3 Eine eigene Fehlerseite

Der schnellste, einfachste, aber nicht unbedingt sauberste Weg ist das Zuweisen einer alternativen Webseite, die angezeigt wird, wenn ein beliebiger Fehler auf der aktuellen Seite auftritt.

Öffnen Sie die Datei *Default.aspx* (HTML-Ansicht) und erweitern Sie gleich die erste Zeile um den Eintrag *ErrorPage*:

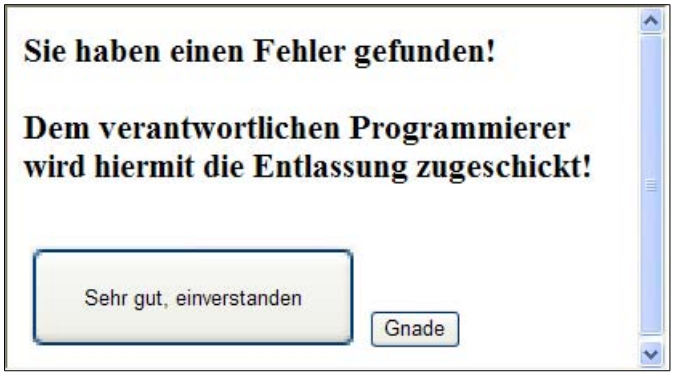


Wählen Sie eine vorhandene Datei aus, oder geben Sie einen anderen Dateinamen an (z.B. *Error.htm*).

Speichern Sie die Datei und erzeugen Sie über den Menüpunkt *Website|Neues Element hinzufügen* eine *Error.htm*-Seite mit einer Fehlermeldung Ihrer Wahl (hierbei handelt es sich um eine reine HTML-Seite, Sie können also – soweit vorhanden – Ihre HTML-Kenntnisse nutzen)[1].

Tritt jetzt ein unbehandelter Fehler auf der Seite auf, wird die eigene Fehlerseite statt der MS-Fehlerseite angezeigt:

[1] Es kann natürlich auch eine ASPX-Datei sein ...



HINWEIS: Sie müssen in der Datei *Web.config* (<*customerrors*>-Tag) das Attribut *mode* auf "On" gesetzt haben, um auf dem Entwicklungsrechner die Seite zu sehen. Anderenfalls bleibt es bei der ausführlichen Fehlermeldung.

42.2.4 Fehlerbehandlung im Web Form

War Ihnen die vorhergehende Lösung etwas zu "billig" und zu wenig aussagekräftig? Vermutlich ja. Wesentlich eleganter lassen sich Fehler im *Page_Error*-Ereignis auswerten. Zum einen stehen Ihnen alle Informationen über Fehlerart, -ort und -meldung zur Verfügung, zum anderen bietet sich Ihnen hier auch die Möglichkeit, weitere Aktionen zu veranlassen. Beispielsweise können Sie dem Entwickler/Webmaster eine ausführliche E-Mail mit den Fehlermeldungen zukommen lassen.

Über das *Server*-Objekt können Sie im *Page_Error*-Ereignis die einzelnen Fehlerdetails abfragen und zum Beispiel über das *Response*-Objekt an den Client senden.

HINWEIS: Nach Ihrer mehr oder weniger ausführlichen Fehlerbehandlung müssen Sie in jedem Fall die Methode *Server.ClearError* aufrufen. Anderenfalls kümmert sich ASP.NET wieder um den Fehler, und das wollten wir ja gerade verhindern.

BEISPIEL 42.22: Anzeige von Fehlerinformationen über das *Page_Error*-Ereignis

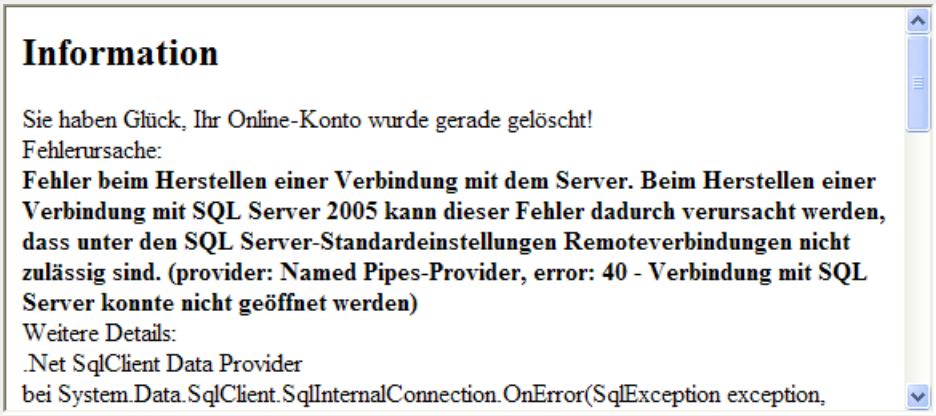
C#

```
private void Page_Error()
{
    Response.Write("<h2>Information</h2>");
    Response.Write("Sie haben Glück, Ihr Online-Konto wurde gerade gelöscht!<br>");
    Response.Write("Fehlerursache:<br>");
    Response.Write("<b>" + Server.GetLastError().Message + "</b><br>");
    Response.Write("Weitere Details:<br>");
    Response.Write(Server.GetLastError().Source + "<br>");
    Response.Write(Server.GetLastError().StackTrace + "<br>");
    Server.ClearError();
}
```

BEISPIEL 42.22: Anzeige von Fehlerinformationen über das *Page_Error*-Ereignis

Ergebnis

Wie erwartet erscheint im Browser die gewünschte Meldung:



HINWEIS: Wie Sie E-Mails (Admin-Benachrichtigung) erzeugen können, zeigt Abschnitt 42.3.

42.2.5 Fehlerbehandlung in der Anwendung

So schön und flexibel die im vorhergehenden Abschnitt vorgestellte Lösung auch sein mag, wer hat schon Lust, für Dutzende Webseiten immer wieder die gleichen Anweisungen einzutippen? Besser ist in diesem Fall eine zentrale Lösung. Wenn Sie jetzt an das *Application*-Objekt denken, liegen Sie gar nicht so falsch.

In der Datei *Global.asax* finden Sie unter anderem bereits die Definition für unseren zentralen Error-Handler, die Methode *Application_Error*. Die Verwendung erfolgt analog zum Ereignis *Page_Error*, die Informationen über den Fehler selbst finden Sie jedoch nicht mehr über *Server.Exception*, sondern über *Server.GetLastError().InnerException*.

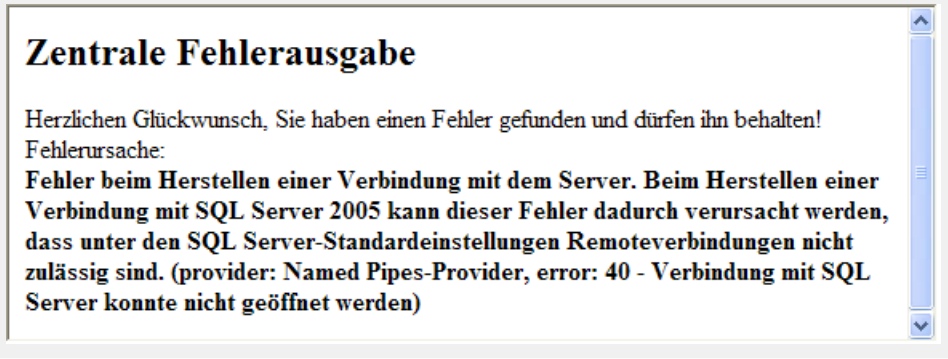
BEISPIEL 42.23: Zentraler Error-Handler

C#

```
void Application_Error(object sender, EventArgs e)
{
    Response.Write("<h2>Zentrale Fehlerausgabe</h2>");
    Response.Write("Herzlichen Glückwunsch, Sie haben einen Fehler gefunden" +
                   " und dürfen ihn behalten!<br>");
    Response.Write("Fehlerursache:<br>");
    Response.Write("<b>" + Server.GetLastError().InnerException.Message + "</b><br>");
    Server.ClearError();
}
```

BEISPIEL 42.23: Zentraler Error-Handler

Ergebnis



Doch Achtung:

HINWEIS: Verwenden Sie gleichzeitig ein *Page_Error*-Ereignis, wird dieses Ereignis zuerst aufgerufen. Löschen Sie hier den Fehler, tritt das Ereignis *Application_Error* nie auf.

42.2.6 Alternative Fehlerseite einblenden

Mit Hilfe der *Web.config* können Sie für den Fall der Fälle eine alternative Seite definieren, die immer dann angezeigt wird, wenn

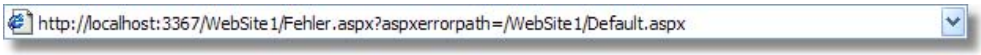
- ein Fehler auftritt (eine Grundvoraussetzung),
- keine *ErrorPage* definiert ist,
- *customErrors mode = "On"* ist.

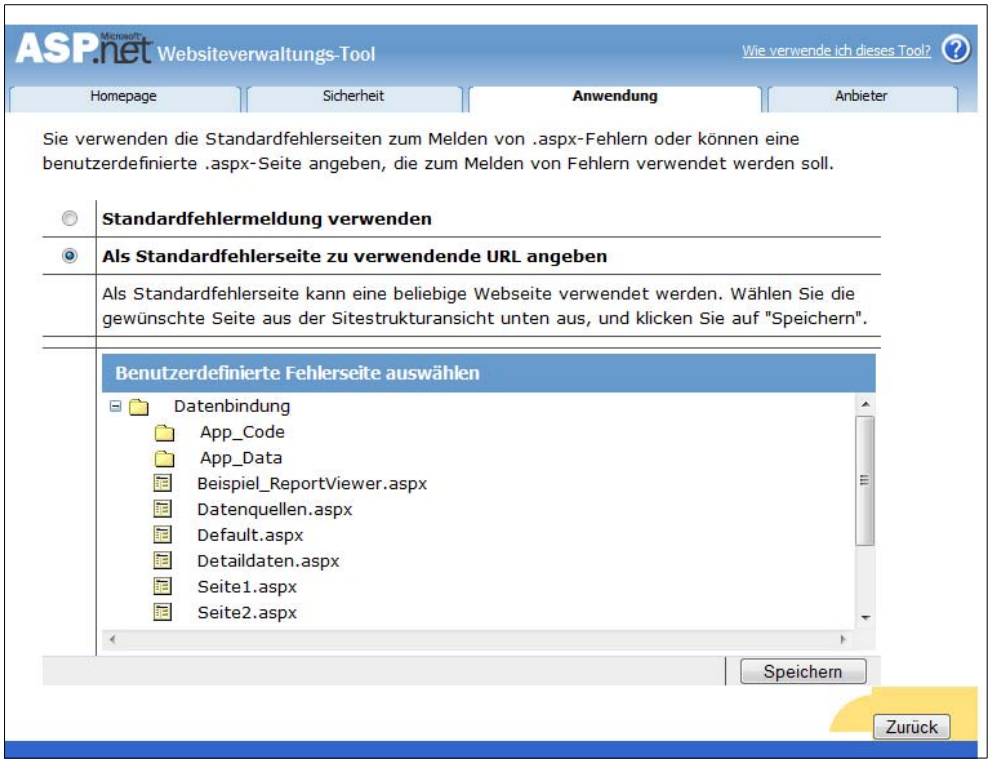
Sie können die Einstellung entweder in der *Web.config* selbst vornehmen

```
<customErrors defaultRedirect="~/Fehler.aspx" mode="On" />
```

oder Sie verwenden die ASP.NET-Konfiguration (siehe folgende Abbildung).

Tritt jetzt ein Fehler zur Laufzeit auf, wechselt ASP.NET automatisch zur festgelegten Seite. Als *QueryString* wird zusätzlich "die Quelle allen Übels" übergeben:





42.2.7 Lokale Fehlerbehandlung

Natürlich müssen Sie es nicht erst so weit kommen lassen, dass eine eigene Fehlerseite notwendig wird. Viele Fehlerursachen sind vorhersehbar und können auch entsprechend abgesichert werden. Dabei stehen Ihnen alle Möglichkeiten von C# zur Verfügung.

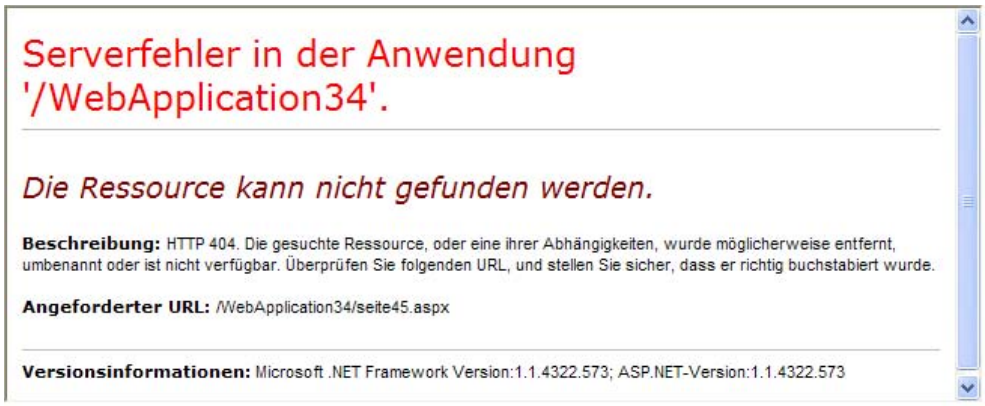
BEISPIEL 42.24: Fehlerbehandlung im *Page_Load*-Ereignis

C#

```
try
{
    this.sqlConnection1.Open();
}
catch
{
   Response.Write("Server nicht verfügbar, versuchen Sie es in einigen Minuten erneut!");
}
```

42.2.8 Seite nicht gefunden – was nun?

Von unzähligen Webseiten ist Ihnen sicher auch die "beliebte" Fehlermeldung mit dem Code 404 (Seite nicht gefunden) in Erinnerung. Die Ursache ist meist der Zahn der Zeit, Seiten werden umbenannt, verschoben oder einfach mal gelöscht, was beim Aufruf dieser Seite dann zu genannter Fehlermeldung durch den IIS führt:



Abhilfe schaffen Sie entweder mit einer zentralen Fehlerbehandlung wie in Abschnitt 42.2.5 beschrieben, oder einer kleinen Erweiterung der Datei *Web.config*[1]:

```
<customErrors mode="On">
   <error statusCode="404" redirect="error.htm" />
</customErrors>
```

HINWEIS: Achten Sie auf die Gross-/Kleinschreibung, anderenfalls erhalten Sie Fehlermeldungen, die Sie sicher nicht anzeigen wollen!

42.3 E-Mail-Versand in ASP.NET

Unvermeidlich und kaum wegzudenken aus der heutigen Zeit ist das Versenden von E-Mails, die teilweise zur echten Plage werden können, sodass nur noch der Wechsel der Mail-Adresse Abhilfe schafft.

Auch in Ihren ASP.NET-Projekten können Sie mit wenigen Zeilen Quellcode zum Beispiel eine Bestellbestätigung, einen E-Mail-Adresstest oder Ähnliches realisieren.

[1] Sie sehen schon, hier handelt es sich um das "Schweizer Taschenmesser" des ASP.NET-Web-Programmierers.

42.3.1 Übersicht

Dreh- und Angelpunkt der weiteren Ausführungen ist die Einbindung des folgenden Namespace, der uns mit den Klassen *SmtpClient* und *MailMessage* beglückt:

```
using System.Net;
```

HINWEIS: Wer bereits mit ASP.NET 1.x gearbeitet hat, wird sich noch an den Namespace *System.Web.Mail* erinnern, in dem sich früher (und auch noch jetzt) die gewünschten Klassen befanden.

Über die Klasse *SmtpClient* bzw. deren Methode *Send* erfolgt der eigentliche Versand. Entweder Sie übergeben alle Parameter einzeln oder Sie definieren vorher ein *MailMessage*-Objekt, das Sie an die Methode übergeben.

SYNTAX: `void Send (string from, string recipients, string subject, string body)`

oder

SYNTAX: `void Send (MailMessage message)`

Einige wichtige Eigenschaften des *MailMessage*-Objekts, das eine einzelne E-Mail darstellt, zeigt die folgende Tabelle:

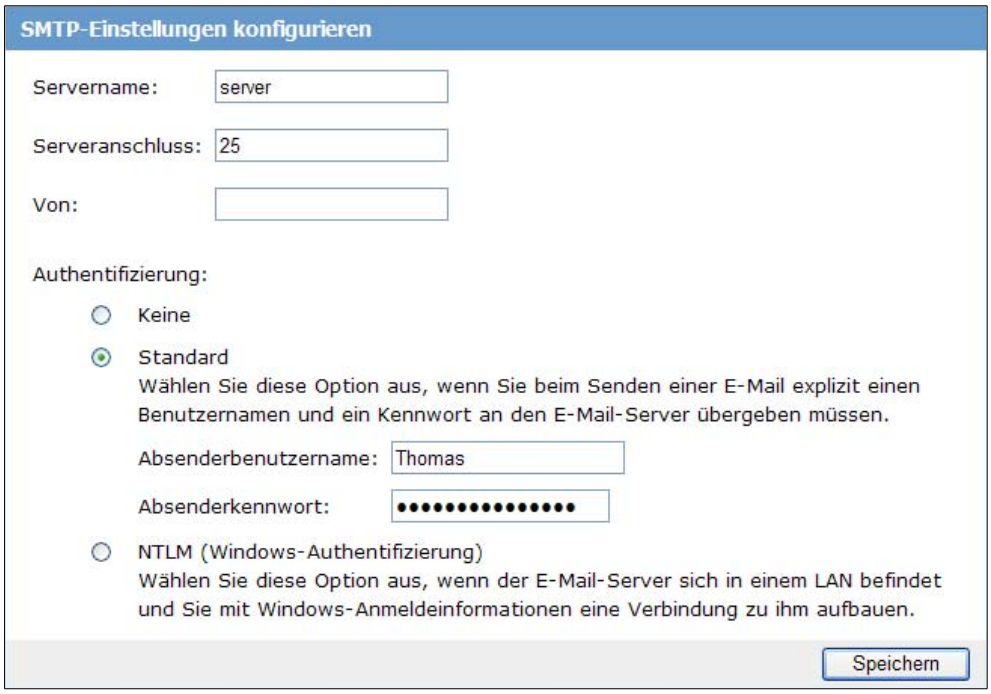
Eigenschaft	Beschreibung
Attachments	… eine Collection aller angehängten Dateien
Bcc	… alle Blind-Copy-Empfänger (mit Semikolon getrennt)
Body	… der Inhalt/Text der E-Mail
BodyFormat	… das Datenformat der E-Mail (*Text* (Standard) oder *HTML*)
Cc	… alle Empfänger (Kopie) der E-Mail (mit Semikolon getrennt)
From	… der Absender
Priority	… die Priorität (*Normal*, *Low*, *High*)
Subject	… die Betreffzeile
To	… die Empfänger (mit Semikolon getrennt)

Die meisten Eigenschaften dürften Ihnen aus eigener Erfahrung bereits bekannt sein.

42.3.2 Mail-Server bestimmen

Wie auch bei jedem Brief sind bei einer E-Mail vor allem Empfänger und Absender interessant, aber wo ist das "Postamt", d h. in unserem Fall der Mail-Server? Meist handelt es sich um einen weiteren PC im Netzwerk, von dem wir zumindest die Adresse und die Einwahldaten kennen sollten.

Die Konfiguration Ihres Mailzugangs nehmen Sie am besten in der *Web.config* mit Hilfe des ASP.NET-Konfigurationsprogramms (*Anwendung|SMTP-Einstellungen*) vor:



Die Einträge in der *Web.config*:

```
...
 <system.net>
  <mailSettings>
   <smtp from="">
    <network host="server" password="geheim" userName="Thomas" />
   </smtp>
  </mailSettings>
 </system.net>
</configuration>
```

Alternativ können Sie die Angaben mit den Eigenschaften des *SmtpClient*-Objekts setzen:

```
System.Net.Mail.SmtpClient mail = new System.Net.Mail.SmtpClient();
...
mail.Credentials = new NetworkCredential("Thomas", "thomas");
mail.Host = "server";
...
```

42.3.3 Einfache Text-E-Mails versenden

Probieren wir es zunächst mit einem simplen Beispiel ohne *MailMessage*-Objekt.

Variante 1

BEISPIEL 42.25: Textnachricht versenden

C#

Fügen Sie einen *Button* in das Web-Form ein und verwenden Sie dessen *Click*-Ereignis zum Absenden der E-Mail.

```
using System.Net;
...
    private void Page_Load(object sender, System.EventArgs e)
    {
        System.Net.Mail.SmtpClient mail = new System.Net.Mail.SmtpClient();
        mail.Host = "localhost";
        mail.Send("th.gewinnus@web.de", "w.doberenz@t-online.de",
            "Das Buch ist fertig!", "Na ja, war nur ein Scherz …");
    }
```

Wie unschwer zu erkennen ist, versendet Thomas Gewinnus eine Mail an Walter Doberenz mit der höchst fragwürdigen Behauptung, dass das Buch fertig sei[1].

Selbstverständlich können Sie auch Controls bzw. deren *Text*-Eigenschaft für die Angabe der Adresse oder des Betrefftextes nutzen:

```
mail.Send("th.gewinnus@web.de", Emfang.Text, Betreff.Text, Inhalt.Text);
```

HINWEIS: Möchten Sie statt der schnöden Adressangaben auch einen Namen verwenden, schreiben Sie diesen einfach vor die E-Mail-Adresse und setzen Sie die E-Mail-Adresse in Größer- bzw. Kleiner-Zeichen.

BEISPIEL 42.26: Zusätzliche Verwendung eines Namens

C#

```
'Thomas Gewinnus<th.gewinnus@web.de>'
```

Variante 2

Wer es gern übersichtlicher und vor allem objektorientierter mag, der erzeugt zunächst ein *MailMessage*-Objekt, dem die einzelnen Eigenschaften zugewiesen werden können.

BEISPIEL 42.27: E-Mail-Versand mit Hilfe des *MailMessage*-Objekts

C#

```
using System.Net;
...
    private void Page_Load(object sender, System.EventArgs e)
```

[1] Jetzt, wo Sie das Buch endlich in den Händen halten, handelt es sich offensichtlich nicht mehr um einen Scherz.

BEISPIEL 42.27: E-Mail-Versand mit Hilfe des *MailMessage*-Objekts

```
    {
        System.Net.Mail.SmtpClient mail = new System.Net.Mail.SmtpClient();
```

Die Adressen können Sie schon beim Erzeugen des *MailMessage*-Objekts angeben:

```
        System.Net.Mail.MailMessage msg = new System.Net.Mail.MailMessage(
                                            "Thomas Gewinnus<th.gewinnus@web.de>",
                                            "w.doberenz@t-online.de");

        msg.Subject = "E-Mail-Test in ASP.NET";
        msg.Body = "Hallo, anbei die fertigen Buchtexte <g>";
        mail.Host = "localhost";
        mail.Send(msg);
    }
```

42.3.4 E-Mails mit Dateianhang

Neben dem reinen Textversand können Sie mit ASP.NET auch Dateien jeglicher Art[1] verschicken. Die Vorgehensweise ist die gleiche wie im vorhergehenden Abschnitt, lediglich der *Attachments*-Collection müssen Sie mit der Methode *Add* ein neu erzeugtes *MailAttachment*-Objekt übergeben. Im Konstruktor wird der jeweilige Dateiname angegeben.

BEISPIEL 42.28: Versand der Datei *Project1.exe*

C#
```
using System.Net;
...
    private void Page_Load(object sender, System.EventArgs e)
    {
        System.Net.Mail.SmtpClient mail = new System.Net.Mail.SmtpClient();
```

Die Adressen können Sie schon beim Erzeugen des *MailMessage*-Objekts angeben:

```
        System.Net.Mail.MailMessage msg = new System.Net.Mail.MailMessage(
                                            "Thomas Gewinnus<th.gewinnus@web.de>",
                                            "w.doberenz@t-online.de");
        msg.Subject = "E-Mail-Test in ASP.NET";
        msg.Body = "Hallo, anbei die fertigen Buchtexte <g>";
        msg.Attachments.Add(new System.Net.Mail.Attachment("c:\\buch.doc"));
        mail.Host = "localhost";
        mail.Send(msg);
    }
```

[1] Leider zählen dazu auch Viren, also Vorsicht!

42.4 Sicherheit von Webanwendungen

Bevor wir uns der eigentlichen Realisierung von Sicherheitskonzepten mit C# bzw. den ASP.NET-Tools zuwenden, ist ein kleiner Ausflug in die Theorie unausweichlich. Das Thema ist allerdings viel zu komplex, um im Rahmen dieses Abschnitts umfassend behandelt werden zu können.

42.4.1 Authentication

Unter *Authentication* (Authentifizierung) verstehen wir das Identifizieren eines Webnutzers aus Sicht der Webanwendung. Unter Windows erfolgt dies automatisch, wenn Sie sich mit einem bestimmten Account, zum Beispiel *Administrator*[1], beim Systemstart anmelden. Nachfolgend können diesem Account bestimmte Rechte eingeräumt oder entzogen werden. Die gleiche Aufgabe besteht auch für eine Webanwendung, nur dass wir es hier mit einer etwas komplizierteren Materie zu tun haben, da Client und Server über ein relativ unsicheres Medium (Internet) miteinander verbunden sind.

Drei sinnvolle Varianten für die Authentifizierung bietet ASP.NET uns an:

- Anonyme Anmeldung
- Forms Authentication
- Windows Authentication

Anonyme Anmeldung

Dies ist die standardmäßige Anmeldung, wie wir sie auch mehr oder weniger unbewusst bei den vorhergehenden Beispielen verwendet haben. Die ASP.NET-Sicherheitsmechanismen sind in diesem Fall weitgehend ausgeschaltet, egal welcher Client auf die Anwendung zugreift, für den IIS handelt es sich

- bei Window XP um den Nutzer *ASPNET,*
- beim IIS 6.x um den Nutzer *NT-AUTORITÄT\NETZWERKDIENST*
- ab dem IIS 7.0 um den Nutzer *ApplicationPoolIdentity* (*IIS APPPOOL\DefaultAppPool*)

Möchten Sie zum Beispiel auf Datenbanken zugreifen oder Dateien erzeugen, müssen Sie diesen Nutzer mit entsprechenden Rechten ausstatten. Beim MS SQL Server muss ein entsprechender Account erzeugt werden, für Access-Datenbanken bzw. andere Dateien/ Verzeichnisse müssen die Schreibrechte (für das Verzeichnis) aktiviert werden.

HINWEIS: Dass Sie mit einer *Anonymen Anmeldung* dem unbefugten Zugriff Tür und Tor öffnen, sollte Ihnen stets bewusst sein!

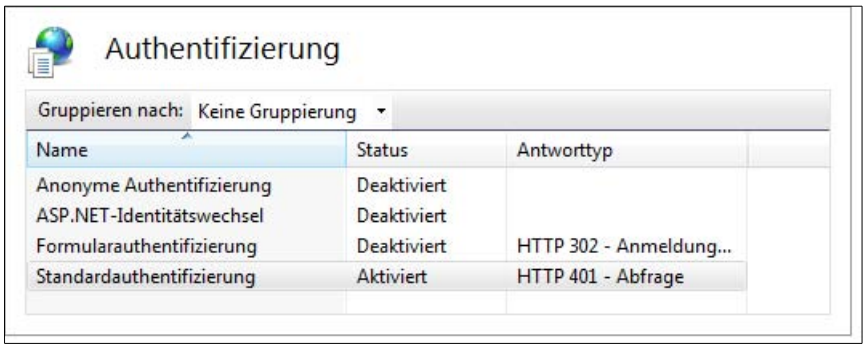
[1] Was nicht unbedingt empfehlenswert ist.

Forms Authentication

Der Name deutet es eigentlich schon an, die Web Forms und damit ASP.NET sind für die Identifizierung des Nutzers zuständig. Rufen Sie ein Formular Ihrer Webanwendung auf, werden Sie zunächst auf eine spezielle Login-Seite umgeleitet. Dies ist der Fall, wenn ASP.NET keinen entsprechenden Cookie vorfindet. Nach der gültigen Eingabe von Name und Passwort wird der bewusste Cookie erzeugt, der den Nutzer beim Zugriff auf weitere Seiten Ihrer Anwendung identifiziert.

Windows Authentication

Hier wird die Authentifizierung dem IIS überlassen. Um einen Eingabedialog wie bei den Web Forms brauchen Sie sich in diesem Fall nicht selbst zu bemühen, dies nimmt Ihnen das System ab. Der Nutzer muss lokal oder in einer NT-Domäne registriert sein. Wie die Identifikation im Detail geschieht, hängt von der Konfiguration des IIS ab (*Systemsteuerung|Verwaltung|Internet Informations Dienste ...*):



Diese Variante der Authentifizierung ist allerdings mit einigen Fallstricken behaftet: So wird entweder Name und Passwort im Klartext übertragen, oder die Authentifizierung wird nicht von allen Web-Browsern unterstützt, oder Sie bekommen Ärger mit dem Zugriff über einen HTTP-Proxy. Das Anwendungsgebiet dürfte sich damit schnell auf ein lokales Intranet beschränken, zumal Sie meist keinen direkten Zugriff/Einfluss auf Windows-Nutzer/-Gruppen haben werden.

Aktivieren Sie diese Authentifizierung über die zentrale *Web.config*-Datei Ihrer Webanwendung:

```
<authentication mode="Windows" />
```

HINWEIS: Eine individuelle Einstellung für einzelne Unterverzeichnisse Ihrer Webanwendung ist nicht möglich!

42.4.2 Forms Authentication realisieren

Nachdem wir festgestellt haben, dass alle Verfahren mit ihren kleinen Problemen behaftet sind, wollen wir uns kurz der Realisierung einer Forms Authentifizierung zuwenden. Diese Variante

bietet dem Programmierer die volle Kontrolle, Sie entscheiden, wie Sie den Nutzer identifizieren (Datenbank, XML-Datei, Active Directory etc.) und ob dieser Login persistent bleibt.

Im Gegensatz zur Realisierung entsprechender Verfahren in ASP brauchen Sie bei ASP.NET nicht für jedes Formular einzeln zu prüfen, ob ein entsprechender Login erfolgt ist (Cookieprüfung), sondern können diese Aufgabe ASP.NET überlassen. Wichtig für Sie sind zwei Schritte:

- Erzeugen eines Login-Dialogs
- Verifikation von Usern und Passwörtern (zum Beispiel über die Datei *Web.config*)

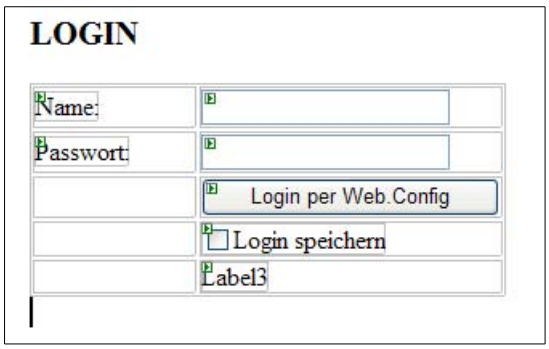
HINWEIS: Möchten Sie den Übertragungsweg (Internet) bei dieser Variante absichern, sollten Sie das Login-Formular per SSL-Verbindung übertragen.

Login-Dialog

Erzeugen Sie zunächst eine neue Anwendung und fügen Sie neben dem Startformular *Default.aspx* noch ein weiteres Formular *Login.aspx* hinzu. Wie der Name schon andeutet, werden wir hier die Authentifizierung des Nutzers implementieren.

Eine entsprechende Vorlage für ein derartiges Formular zeigt die folgende Abbildung.

HINWEIS: Bitte verwenden Sie nur Controls aus der Palette "Standard".



Die *Visible*-Eigenschaft von *Label3* setzen Sie auf *False*, hier werden wir bei Bedarf eine Fehlermeldung einblenden.

HINWEIS: Die *CheckBox* am unteren Rand ist optional. Will der Nutzer einen Cookie mit den Anmeldedaten speichern, so braucht er nicht bei jedem Start der Anwendung seine Credentials neu einzugeben.

Mit dem Klick auf den Button wird es ernst:

```
protected void Button1_Click(object sender, EventArgs e)
{
```

Prüfen von Name und Passwort:

```
if (FormsAuthentication.Authenticate(TextBox1.Text, TextBox2.Text))
```

Alles o k., dann geht es zurück zur Startseite (soll der Login gespeichert werden, muss dies als zweiter Parameter übergeben werden):

```
            FormsAuthentication.RedirectFromLoginPage(TextBox1.Text, CheckBox1.Checked);
```

Fehler beim Login:

```
        else
        {
            Label3.Visible = true;
            Label3.Text = "Fehlerhafte Anmeldung";
        }
    }
```

Damit ist *Login.aspx* fertig. In *Default.aspx* werden wir noch den Namen des aktiven Nutzers ausgeben. Dazu erweitern wir das *Page_Load*-Ereignis:

```
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.Write(User.Identity.Name);
    }
```

Das waren zunächst die Änderungen an der Anwendung selbst, wenden wir uns nun der Konfiguration zu.

Die Änderungen in der Web.config

Öffnen Sie die Datei *Web.config*, und editieren Sie das Attribut *Mode*:

```
  <authentication mode="Forms">
```

Hier legen Sie den Namen des Login-Dialoges fest:

```
    <forms loginUrl="login.aspx" >
```

Das Passwort ist im Klartext gespeichert (alternativ können Sie auch einen Hash-Wert angeben):

```
        <credentials passwordFormat="Clear">
```

Ein erster Eintrag für Username und Passwort:

```
            <user name="HansWurst" password="geheim"/>
        </credentials>
    </forms>
  </authentication>
```

HINWEIS: Mit der folgenden Authorization-Definition wird der unautorisierte Zugriff auf alle Seiten Ihrer Anwendung verhindert. Nur so wird auch *Login.aspx* angezeigt.

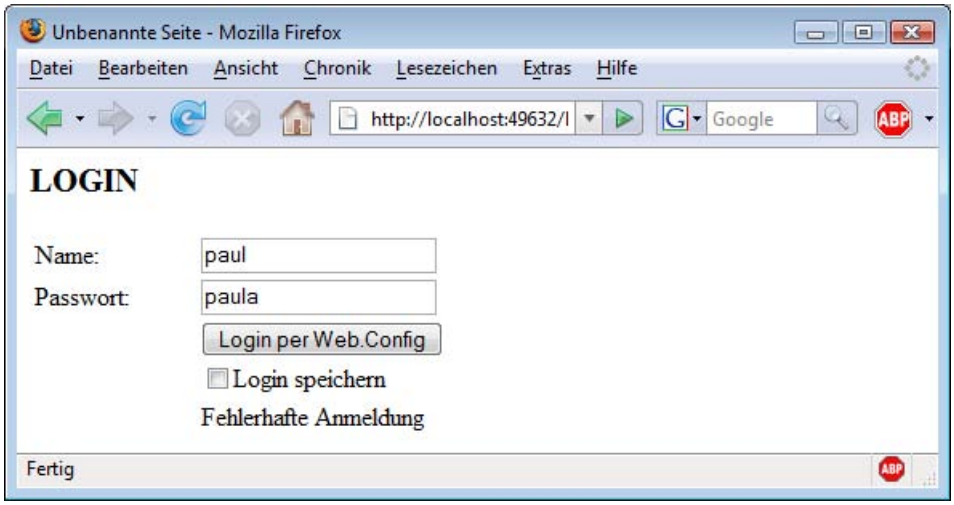
```
<authorization>
    <deny users="?" />
</authorization>
```

HINWEIS: Sie können später einzelne Seiten von der Authentication ausschließen.

Test

Nach dem Start der Anwendung (zunächst findet sich in der Adresszeile auch *Default.aspx*) landen Sie unweigerlich auf der Login-Seite *Login.aspx*. Wohin es im Erfolgsfall zurück geht, zeigt uns ein Blick in die Adresszeile.

Probieren Sie ruhig einmal einen falschen Account aus. Sie landen unweigerlich wieder im Login-Dialog.



Ist die Anmeldung erfolgreich, brauchen Sie sich im Verlauf der Anwendung nicht weiter um Identitätsprüfungen zu kümmern. Über *User.Identity.Name* können Sie später jederzeit den aktuellen Nutzer identifizieren und zum Beispiel Eingabefelder ausblenden.

Haben Sie das persistente Speichern des Logins aktiviert (zweiter Parameter bei *RedirectFromLoginPage*), bestimmt das Attribut *timeout* in der *Web.config*, wie lange dieses Cookie aktuell ist:

```
<authentication mode="Forms">
<forms loginUrl="login.aspx" timeout="120">
  <credentials passwordFormat="Clear">
    <user name="HansWurst" password="geheim"/>
  </credentials>
</forms>
</authentication>
```

Weitere Möglichkeiten der Authentifizierung

Neben der im vorhergehenden Beispiel demonstrierten Variante, mit

```
    if (FormsAuthentication.Authenticate(TextBox1.Text, TextBox2.Text))
```

einen Nutzer über die Datei *Web.config* zu identifizieren, bietet sich auch eine ganz simple Realisierung mittels Textvergleich an:

```
    if ((TextBox1.Text == "Maier")&(TextBox2.Text == "Esel"))
        FormsAuthentication.RedirectFromLoginPage(TextBox1.Text, CheckBox1.Checked);
...
```

Wesentlich eleganter ist natürlich eine Abfrage in einer Datenbank, was sicher bei einer größeren Nutzerzahl und eventuell vorhandenen Anmeldeformularen praktikabler ist.

Doch bevor Sie an dieser Stelle Ihre ADO.NET-Kenntnisse zum Einsatz bringen, möchten wir davon abraten, da es für diesen Fall bereits ein ausgereifte Lösung gibt, die wir Ihnen ab Abschnitt 42.4.6 vorstellen möchten.

42.4.3 Impersonation

Wie bereits mehrfach erwähnt, wird standardmäßig der Benutzer-Account *ASPNET* bzw. *NT-AUTORITÄT\NETZWERKDIENST* bzw. *IIS APPPOOL\DefaultAppPool* durch den IIS verwendet (siehe Seite 2071). Dies ist jedoch nicht in allen Fällen erwünscht. Soll dem IIS bzw. dem zugrunde liegenden Betriebssystem ein anderer Nutzer vorgegeben werden, nennt man dies Impersonation. Der ASP.NET-Engine stehen damit alle Ressourcen des betreffenden Accounts zur Verfügung. Doch Achtung:

HINWEIS: ASP.NET ist durch die Impersonation auch an die Einschränkungen des betreffenden Accounts gebunden.

So ist es unbedingt erforderlich, dass Schreib- und Leserechte für das folgende Verzeichnis eingeräumt werden, anderenfalls können die ASPX-Dateien nicht kompiliert werden:

```
<WINDOWS>\Microsoft.NET\Framework\v4.xxxx\Temporary ASP.NET Files
```

Ein Verwendungszweck für Impersonation bietet sich zum Beispiel beim Zugriff auf den MS SQL Server an. Im Normalfall dürfte der ASP.NET-Nutzer keinen Zugriff haben[1], per Impersonation könnten Sie jedoch mit einer vertrauten Verbindung (ASP.NET nutzt jetzt einen lokalen Account) die gewünschten Daten abrufen.

Drei Varianten der Impersonation sind möglich:

- Automatische Impersonation
- Statische Impersonation
- Dynamische Impersonation

[1] Es sei denn, der SQL Server-Admin hat diesen Account hinzugefügt, was nicht empfehlenswert ist.

42.4.4 Authorization

In den beiden vorhergehenden Abschnitten hatten wir uns damit beschäftigt, einen Web-User zu identifizieren bzw. der ASP.NET-Engine beizubringen, eine neue Identität zu verwenden. Beide Verfahren sind natürlich kein Selbstzweck, sondern dienen als Grundlage für den nächsten Schritt, die Autorisierung.

Aufgabe der Authorization (Autorisierung) ist die Freigabe bzw. Zuordnung von Ressourcen und Rechten für den aktiven User. Dabei werden zwei Arten, die unabhängig voneinander arbeiten, unterschieden:

- URL Authorization
- File Authorization

Eingehende Clientanfragen werden zunächst durch ASP.NET per URL-Authorization geprüft. Grundlage dieser Prüfung ist die angeforderte URL, daher auch die Bezeichnung.

Nach diesem Schritt wird die Anfrage auf Basis der Systemfreigaben (Ordner/Dateien) auf Benutzer- oder Gruppenebene durch Windows überprüft. Grundlage der Prüfung ist die so genannte Access Control List (ACL).

File Authorization

Mit dieser Form der Autorisierung haben Sie unter Windows sicher schon Bekanntschaft geschlossen. Es handelt sich um nicht anderes als die Freigabe von Dateien und Ordnern für bestimmte Nutzer oder Gruppen. ASP.NET agiert quasi als normaler Nutzer (Standard ASPNET/NETZWERKDIENST) und ist damit natürlich auch an die entsprechenden Restriktionen gebunden.

Ein ganz praktisches Anwendungsgebiet ist der Zugriff auf Access-Datenbanken (Schreibzugriff), wenn sich diese in einem Webordner befinden. Per Defaulteinstellung kann zwar *Jeder* auf diesen Ordner **lesend** zugreifen, eine Berechtigung für das Schreiben müssen Sie jedoch erst explizit vornehmen.

Ganz anders der Fall, wenn Sie per Impersonation zum Beispiel den Admin-Account missbrauchen. Jetzt agiert ASP.NET als Administrator und hat natürlich auch alle Rechte.

URL Authorization

URL Authorization wird per *Web.config* realisiert und ist damit lediglich für ASP.NET relevant. Über das *Authorization*-Element können Sie definieren, welcher Benutzer Zugriff auf die Web-Applikationen erhält und welcher nicht.

BEISPIEL 42.29: Ein erstes Beispiel hatten Sie ja schon im Zusammenhang mit der Forms Authentication kennen gelernt:

XML

```
<authentication mode="Forms">
<forms loginUrl="login.aspx" timeout="120">
  <credentials passwordFormat="Clear">
    <user name="HansWurst" password="geheim"/>
```

BEISPIEL 42.29: Ein erstes Beispiel hatten Sie ja schon im Zusammenhang mit der Forms Authentication kennen gelernt:

```
    </credentials>
  </forms>
  </authentication>

  <authorization>
   <deny users="?" />
  </authorization>
```

Mit <*deny*> verwehren Sie den Zugriff, das Fragezeichen steht für alle anonymen Benutzer. Da wir alle anonymen User ausgeschlossen haben, besteht ein Zwang zur Anmeldung (in diesem Fall Web Forms).

HINWEIS: Neben dem Fragezeichen gibt es noch den "*" als Platzhalter. In diesem Fall handelt es sich um alle Nutzer, ob angemeldet oder nicht.

Neben Platzhaltern können Sie auch direkt die Namen von Usern oder auch ganzen Gruppen für das Zuweisen bzw. das Entziehen von Rechten nutzen. Geben Sie für die Attribute *users* und *roles* einfach eine kommagetrennte Liste der Namen an.

BEISPIEL 42.30: Alle Administratoren sowie Maier und Schulze erhalten vollen Zugriff.

XML
```
...
  <authorization>
     <allow verb="GET,POST,HEAD,DEBUG"
        users="Maier,Schulze"
        roles="Administrators"/>
     <deny users="*"/>
  </authorization>
...
```

Im Gegensatz zur *Authentication* ist die *Authorization* auch auf bestimmte Unterverzeichnisse beschränkbar, was sicher auch einen Sinn macht. So können Sie zwar das gesamte Web freigeben, für bestimmte Ordner sind jedoch eine Anmeldung und ein bestimmter Account erforderlich. Verantwortlich für diese Vielfalt ist das <*location*>-Element.

BEISPIEL 42.31: Zugriff auf das Verzeichnis *Download*

XML
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

Grundsätzlich freier Zugriff:

```
 <system.web>
  <authentication mode="None"/>
  <authorization>
   <allow users="*"/>
```

BEISPIEL 42.31: Zugriff auf das Verzeichnis *Download*

```
  </authorization>
 </system.web>
```

Aber für *Download* gelten Einschränkungen und die Pflicht zur Anmeldung:

```
 <location path="download"
  <system.web>
   <authentication mode="Forms">
    <forms loginUrl="/login.aspx"/>
   </authentication>
   <authorization>
    <allow users="Maier"/>
    <deny users="*"/>
   </authorization>
  </system.web>
 </location>
</configuration>
```

42.4.5 Administrieren der Website

Wie schon im Abschnitt 42.4.1 besprochen, bietet ASP.NET seit Version 2.0 einen reichhaltigen Fundus an Funktionen zur Forms-basierten Authentifizierung:

- eine komfortable Web-Oberfläche (ASP.NET-Konfiguration),
- diverse Objekte und Klassen zur Administration per Code
- sowie diverse Server-Steuerelemente für die Authentifizierung und Administration der Sicherheitsfeatures.

Voraussetzung

HINWEIS: Grundvoraussetzung ist derzeit die Installation eines *ASPNetSqlProviders*, d h. eines SQL Servers. Dieser Provider ermöglicht das Speichern der Nutzerdaten und -rechte in einer Datenbank des jeweiligen Providertyps.

Mit der Installation von Visual Studio wird in der Regel auch ein *SQL Server Express* installiert. Dieser ist per Default als *ASPNetSqlProvider* in der *Machine.config* eingetragen:

```
...
<providers>
   <add name="AspNetSqlMembershipProvider" type="System.Web.Security.SqlMembershipProvider,
        System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
        connectionStringName="LocalSqlServer" enablePasswordRetrieval="false"
        enablePasswordReset="true" requiresQuestionAndAnswer="true" applicationName="/"
        requiresUniqueEmail="false" passwordFormat="Hashed" maxInvalidPasswordAttempts="5"
        minRequiredPasswordLength="7" minRequiredNonalphanumericCharacters="1"
```

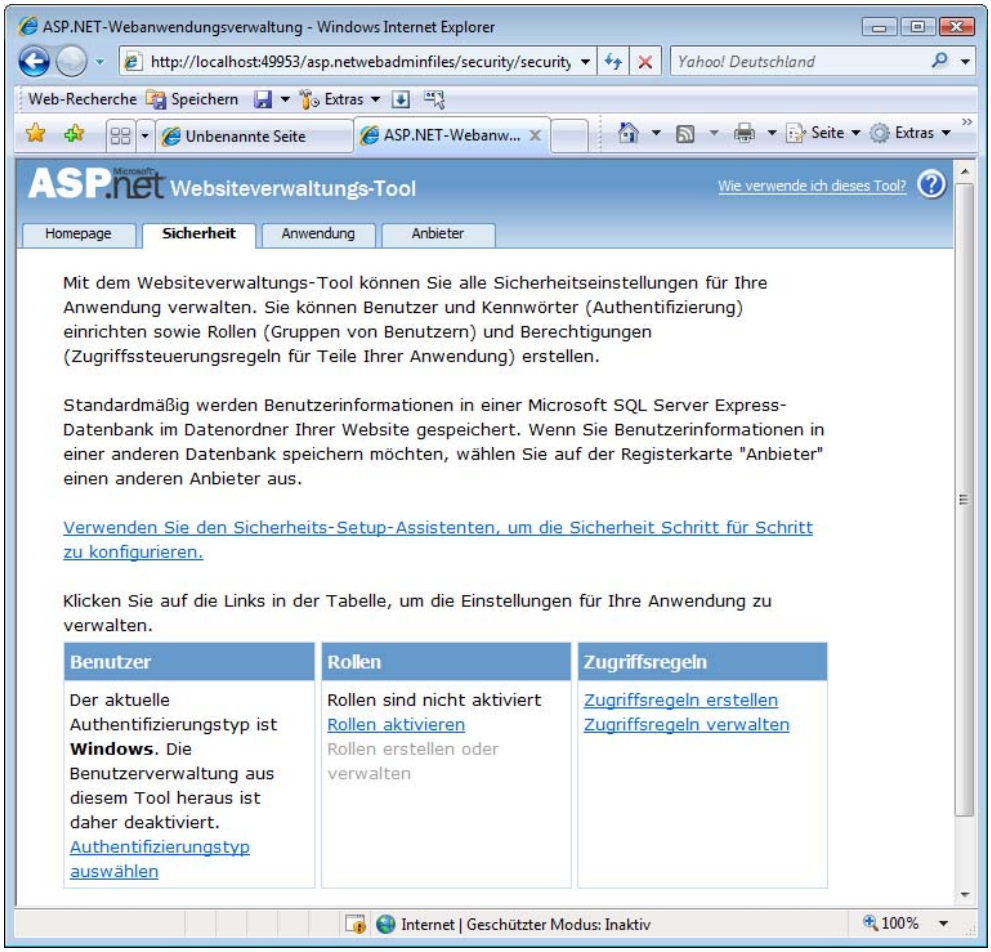
```
        passwordAttemptWindow="10" passwordStrengthRegularExpression="" />
</providers>
```

Hier findet sich auch die Information für den Ablageort der Daten:

```
<connectionStrings>
  <add name="LocalSqlServer" connectionString="data source=.\SQLEXPRESS;Integrated
       Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb mdf;User Instance=true"
       providerName="System.Data.SqlClient" />
</connectionStrings>
```

ASP.NET-Konfigurationsoberfläche

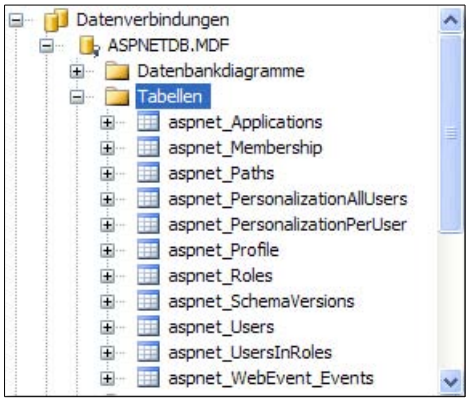
Über den Menüpunkt *Website|ASP.NET-Konfiguration* rufen Sie eine Administrationsoberfläche für Ihre Web-Anwendung auf. Diese verändert zum einen Einstellungen in der *Web.config*, zum anderen wird hier ggf. auch die Datenbank mit den Usern und deren Rechten verwaltet.



Auswahl des Authentifizierungstyps

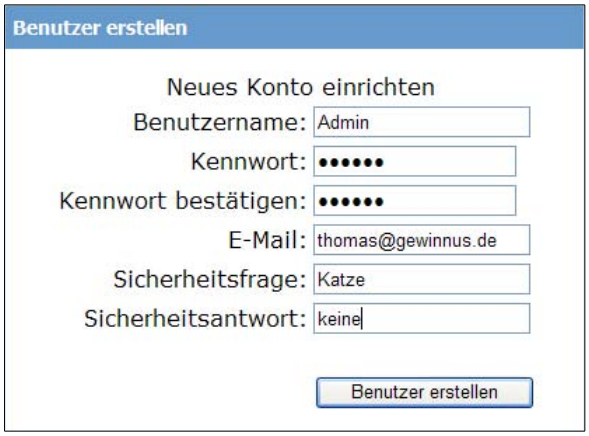
Erster Schritt in der Rubrik *Sicherheit* ist die Auswahl des Authentifizierungstyps (gut, dass wir darüber schon gesprochen haben). Sie haben die Wahl zwischen *Internet* und *lokalem Netzwerk*, zu deutsch: zwischen Forms-basierter Authentifikation und Windows-basierter Authentifikation. Im Allgemeinen werden Sie wohl für die Forms-basierte Authentifikation entscheiden.

Mit Klick auf den entsprechenden Menüpunkt wird automatisch eine *ASPNETDB.MDF* im *App_Data*-Verzeichnis erzeugt. Diese SQL Server-Datenbank verwaltet im Weiteren die Nutzer, deren Einstellungen, Rollen, Zugriffsregeln etc. Ein Blick in die Datenbank zeigt bereits jede Menge Tabellen in denen diese Daten abgelegt werden sollen:



Neuen Nutzer erstellen

Wie wohl nicht anders zu erwarten, fängt alles mit dem Erstellen eines neuen Nutzers[1] an:



[1] Zum Nachvollziehen des Buchbeispiels: Passwort="Administrator!"

Wofür die E-Mail? Wer sich schon bei diversen Webshops angemeldet hat, wird diese Option zu schätzen wissen, besteht doch so die Möglichkeit, ein vergessenes Passwort zugesendet zu bekommen. Abgesichert wird dieses Verfahren mit der Sicherheitsfrage.

HINWEIS: Achtung: Ihr Passwort muss gewissen Mindeststandards genügen, anderenfalls "meckert" ASP.NET (Minimum-Länge 6 Zeichen).

Rechte einschränken

Starten Sie Ihre Anwendung mit diesen Einstellungen passiert zunächst nichts. Wer den Abschnitt bis hierher aufmerksam gelesen hat, wird nicht überrascht sein, ist doch in der *Web.config* bisher noch keine Einschränkung vorgenommen worden.

Erstellen Sie deshalb in der Konfigurationsoberfläche noch eine neue Zugriffsregel und schließen Sie zunächst alle *Anonymen Benutzer* aus:

Die *Web.config* nach diesen Einstellungen per Konfigurationsoberfläche:

```
<?xml version="1.0"?>
<configuration>
 <system.web>
  <authorization>
   <deny users="?" />
  </authorization>
  <authentication mode="Forms"/>
   <compilation debug="true"/></system.web>
</configuration>
```

Der erneute Versuch, die Anwendung zu starten, dürfte jetzt fehlschlagen, da wir noch keine Anmeldung realisiert haben.

HINWEIS: Aus mehreren Nutzern können Sie Rollen bilden, die wiederum mit Rechten versehen werden können. Diese Option muss zunächst per Konfigurationsoberfläche aktiviert werden, da standardmäßig keine Rollen zugelassen sind.

42.4.6 Steuerelemente für das Login-Handling

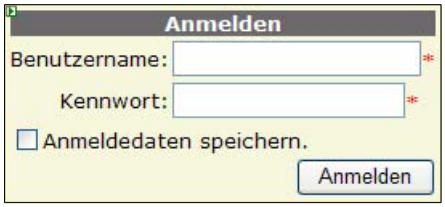
Nachdem wir die Website für anonyme Nutzer gesperrt haben, müssen wir eine Möglichkeit schaffen, dass sich ein Nutzer auch anmelden kann. Dazu stellt ASP.NET mehrere Server-Controls zur Verfügung (Rubrik *Anmelden*):

- *Login*
- *LoginView*
- *PasswortRecovery*
- *LoginStatus*
- *LoginName*
- *CreateUserWizard*
- *ChangePasswort*

Die Bedeutung der einzelnen Steuerelemente erschließt sich teilweise bereits aus dem Namen, für weitere Klarheit sollen die folgenden Beispiele sorgen.

Login

Dieses Controls ermöglicht die Authentifizierung des Nutzers mit dessen Name und Passwort:



Fügen Sie das Control am besten in eine eigene Seite ein (Default *Login.aspx*).

Über die Eigenschaft *DisplayRememberMe* bestimmen Sie die Anzeige der CheckBox im obigen Dialog und damit auch, ob sich Anmeldedaten speichern lassen.

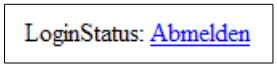
Die späteren, mehr oder weniger freundlichen, Hinweise der Dialogbox können Sie über die Eigenschaften *FailureText*, *PasswordRequiredErrorMessage* und *UserNameRequiredErrorMessage* beeinflussen:

Haben Sie das Control in einer normalen ASPX-Seite untergebracht, können Sie es mit der Eigenschaft *VisibleWhenLoggedIn* automatisch ausblenden, wenn die Anmeldung erfolgreich war.

Einem Test der Anwendung steht jetzt nichts mehr im Weg, geben Sie "Admin" und "Administrator!" ein, sollten Sie erfolgreich angemeldet werden.

LoginStatus

Dieses recht einfache Control zeigt Ihnen zum einen den aktuellen Anmeldestatus an, zum anderen handelt es sich um einen Hyperlink, mit dem Sie sich auch an-/abmelden können:



LoginName

Noch einfacher als *LoginStatus* ist das Control *LoginName* aufgebaut. Es zeigt lediglich den Namen des aktuellen Nutzers an.

BEISPIEL 42.32: LoginName

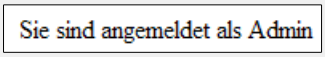
C#

Über die Eigenschaft *Formatstring* können Sie neben dem Platzhalter ({0}) noch zusätzlichen Text eingeben.

```
Sie sind angemeldet als {0}
```

Ergebnis

Später erhalten Sie dann zum Beispiel die folgende Anzeige:



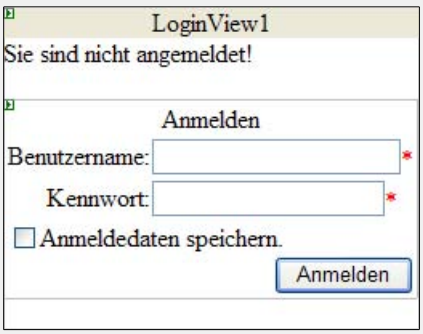
LoginView

Dieses Control zeigt in Abhängigkeit von der Anmeldung zwei verschiedene Templates an:

- *AnonymousTemplate* (keine Anmeldung)
- *LoggedInTemplate*

Sie können also beispielsweise einen Anmeldedialog anzeigen, wenn der Nutzer noch nicht authentifiziert ist, oder bestimmte Nutzerdaten, wenn er sich angemeldet hat.

BEISPIEL 42.33: AnonymousTemplate

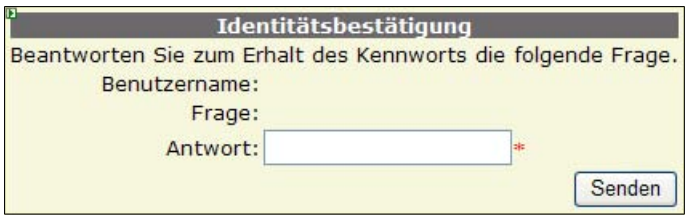


PasswortRecovery

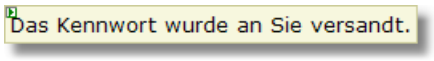
Der Mensch ist vergesslich und auch der normale Computernutzer ist natürlich ein Mensch. Aus diesem guten Grund findet sich im Werkzeugkasten eine "Gedächtnisstütze" in Form des *PasswordRecovery*-Controls:

Da es sich um einen Assistenten mit mehreren Schritten handelt, sind noch zwei weitere Templates vorhanden.

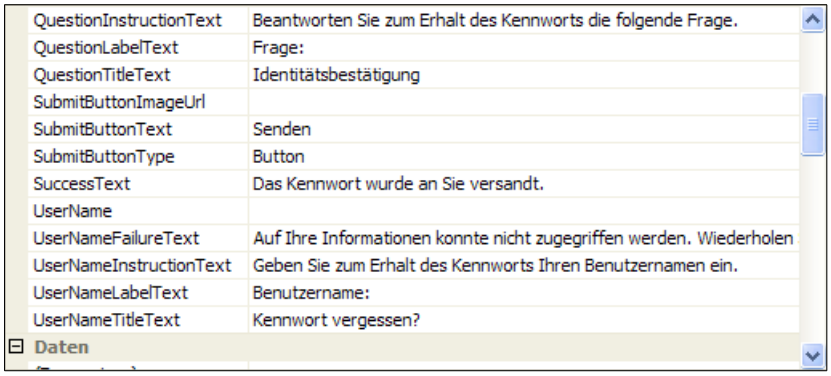
Nach der Eingabe des Benutzernamens muss der Vergessliche eine Frage beantworten (hoffentlich hat er die Antwort nicht auch vergessen):



Wurde die Frage korrekt beantwortet, kommt noch eine kurze Bestätigung:



Auf alle Beschriftungen etc. können Sie natürlich über die zur Verfügung stehenden Eigenschaften Einfluss nehmen:



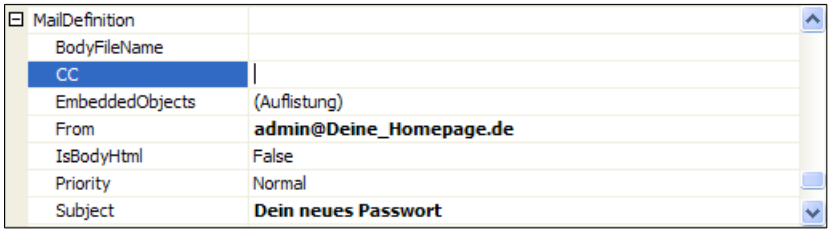
HINWEIS: Wem das nicht reicht, der kann auch das Template in eine Vorlage umwandeln und dann frei über die Gestaltung des Controls entscheiden.

Das Control kümmert sich automatisch um

- das Generieren eines neuen Passworts
- und den Versand der E-Mail (der Mailserver muss natürlich konfiguriert sein).

Warum ein neues Passwort? Die Antwort findet sich in der Datenbank. Dort werden die Nutzerpasswörter verschlüsselt (gehasht) abgelegt. Das ursprüngliche Passwort kann aus dem Hashwert nicht wieder hergestellt werden, deshalb muss ein neues Passwort generiert werden.

Bitte vergessen Sie nicht, die Mail-Konfiguration (Eigenschaft *MailDefinition*) vorzunehmen:

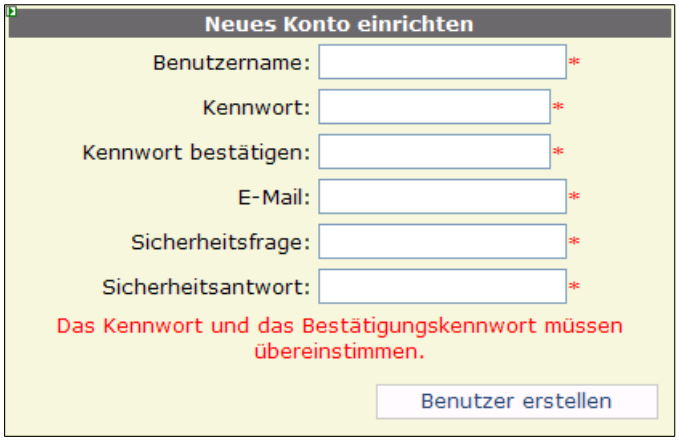


Neben Betreff (*Subject*) und Absender (*From*) können Sie auch noch einen längeren Text (*BodyFileName*) definieren.

CreateUserWizard

Der Mensch ist nun mal faul und so will auch die Neuanmeldung von Nutzern automatisiert sein. Eine Hilfe ist dabei der *CreateUserWizard* (ein Server Control).

Die Bezeichnung "Wizard" ist eigentlich übertrieben, gibt es doch nur noch einen weiteren Schritt, der das erfolgreiche Erzeugen des Kontos bestätigt. Allerdings ist es möglich, weitere Schritte hinzuzufügen.



ChangePasswort

Ist der Nutzer seines Passworts überdrüssig, kann mit dem *ChangePassword*-Control diesem Leiden abgeholfen werden:

HINWEIS: Voraussetzung für die Verwendung des Controls durch den Nutzer ist natürlich die vorherige Anmeldung.

42.4.7 Programmieren der Sicherheitseinstellungen

Was mit der ASP.NET-Konfigurationsoberfläche und den vorgestellten ASP.NET-Controls möglich ist, muss sinnvollerweise auch per Quellcode realisierbar sein.

Schnittstelle für die gewünschte Funktionalität sind die folgenden Klassen:

- *Membership*
- *MembershipUser*
- *Roles*

Um das Thema nicht ganz ausufern zu lassen, beschränken wir uns im Weiteren auf einige Beispiele zum *Membership*-Objekt.

BEISPIEL 42.34: Vorhandenen Nutzer anmelden

C#
```
protected void Button1_Click(object sender, EventArgs e)
{
    if (Membership.ValidateUser("Admin", "Administrator!"))
        FormsAuthentication.RedirectFromLoginPage("Admin", true);
}
```

BEISPIEL 42.35: Informationen über den aktuellen Nutzer abrufen

C#
```
ListBox1.Items.Add("Nutzername = " + Membership.GetUser().UserName);
ListBox1.Items.Add("Nutzer seit = " + Membership.GetUser().CreationDate.ToString());
ListBox1.Items.Add("EMail = " + Membership.GetUser().Email);
ListBox1.Items.Add("Online = " + Membership.GetUser().IsOnline.ToString());

ListBox1.Items.Add("Letzte Aktivität = " +
      Membership.GetUser().LastActivityDate.ToString());

ListBox1.Items.Add("Letztes Logout = " +
      Membership.GetUser().LastLockoutDate.ToString());

ListBox1.Items.Add("Letztes Login = " +
      Membership.GetUser().LastLoginDate.ToString());
ListBox1.Items.Add("Passwortfrage = " + Membership.GetUser().PasswordQuestion);
```

Ergebnis

Die Ausgabe:

```
Nutzername = Admin
Nutzer seit = 05.12.2005 11:34:18
EMail = thomas@gewinnus.de
Online = True
Letzte Aktivität = 05.12.2005 14:52:46
Letztes Logout = 01.01.1754 01:00:00
Letztes Login = 05.12.2005 14:32:35
Passwortfrage = Katze
```

BEISPIEL 42.36: Einen neuen Nutzer anlegen

C#
```
MembershipCreateStatus status = new MembershipCreateStatus();
Membership.CreateUser("Mayer", "geheim_21", "mayer@mayer.de", "gehalt", "keins",
                      true, out status);
```

BEISPIEL 42.37: Einen Nutzer löschen

C#
```
Membership.DeleteUser("Mayer", true);
```

BEISPIEL 42.38: Alle Nutzer anzeigen

C#
```
ListBox2.DataSource = Membership.GetAllUsers();
ListBox2.DataBind();
```

BEISPIEL 42.38: Alle Nutzer anzeigen

oder

```
foreach (MembershipUser _user in Membership.GetAllUsers())
    ListBox2.Items.Add(_user.UserName);
```

BEISPIEL 42.39: Anzahl der eingeloggten User bestimmen

C#
```
Label1.Text = Membership.GetNumberOfUsersOnline().ToString();
```

Damit dürften Sie einen ersten Eindruck darüber gewonnen haben, wie Sie auch per Quellcode Einfluss auf die Nutzerverwaltung nehmen können.

42.5 AJAX in ASP.NET-Anwendungen

Nachdem Sie sich bereits durch mehrere Kapitel zum Thema "ASP.NET" hindurch gequält haben, kommt hier ein weiteres Schlagwort: AJAX.

42.5.1 Was ist AJAX und was kann es?

Bei AJAX handelt sich nicht etwa um ein Konkurrenzprodukt zu ASP.NET und auch nicht um eine gänzlich neue Technologie von Microsoft, sondern schlicht um die sinnvolle Verknüpfung bekannter Technologien (*HTML, DOM, JavaScript, XmlHttpRequest ...*) mit dem Ziel, einige der bisher bestehenden Einschränkungen in der Webprogrammierung zu umgehen. Dabei ist AJAX nicht an eine Microsoft-Technologie wie ASP.NET gebunden, sondern **kann auch** in ASP.NET-Anwendungen zum Einsatz kommen[1].

Zunächst wollen wir es nicht versäumen, Ihnen die Übersetzung des Begriffs nahezubringen, AJAX steht für *Asynchronous JavaScript and XML*, was bereits einige Rückschlüsse auf das Prinzip zulässt.

Grundprinzip jeder konventionellen Webanwendung ist der dauernde Datenaustausch zwischen Client (Browser) und Server (im Fall von ASP.NET der IIS). Nutzeraktivitäten am Client (Klick auf Schaltfläche etc.) lösen eine Serveranfrage aus, die je nach Verbindungsqualität mehr oder weniger schnell beantwortet wird und zu einer neuen Darstellung im Browser führt.

Sicher ist Ihnen beim Experimentieren mit ASP.NET-Anwendungen schon aufgefallen, dass dieser Datenaustausch mitunter recht langsam ist und auch zu mehr oder weniger unschönen optischen Effekten führen kann. Last but not least, vermissen Sie in Webanwendungen wahrscheinlich auch jede Menge von Ereignissen, deren Nutzung sich durch den langsamen Datenaustausch allerdings von selbst verbietet (Mausereignisse, KeyDown etc.).

Genau diese Lücke soll AJAX füllen, indem dem Client wesentlich mehr Aufgaben zugeteilt werden als bisher. Doch genau dafür braucht man zunächst eine gemeinsame Programmierplatt-

[1] Allein zu "AJAX" gibt es mittlerweile mehr Bücher als Sand am Meer. Die Autoren werden deshalb nicht der Versuchung erliegen, dieses komplexe Thema in epischer Breite darzustellen.

form, die von den meisten Browsern unterstützt wird. Kleinster gemeinsamer Nenner ist deshalb JavaScript, dessen Verfügbarkeit auf dem Client auch gleich einer der Knackpunkte von AJAX ist[1].

Aufgabe der in JavaScript programmierten AJAX-Bibliotheken auf dem Client ist die Übernahme möglichst vieler Aufgaben unabhängig vom Server, d.h., ohne Datenaustausch mit diesem. Werden Daten vom Server benötigt, überträgt die AJAX-Library diese asynchron im Hintergrund (per *XmlHttpRequest*-Objekt), der Nutzer sollte mit der Oberfläche weiterarbeiten können. Die AJAX-Library ist weiterhin in der Lage, Teile des Browserinhalts dynamisch neu aufzubauen, ohne einen kompletten Seitenrefresh durchzuführen.

Aus der Verwendung von AJAX ergeben sich damit folgende Vorteile:

- Verbesserte Performance von Webanwendungen, da nicht immer alle Teile der Weboberfläche neu gerendert/übertragen werden müssen.
- Webanwendungen können wesentlich komfortablere Controls verwenden, die über erweiterte Funktionalität auf dem Client verfügen (Progressbars, Tooltips, Popups ...).
- Unterstützung von Forms Authentication und Nutzerprofilen.
- Es besteht die Möglichkeit, über einen Proxy Methoden von Webdiensten aufzurufen.

Leider ist obige Funktionalität nicht zum Nulltarif und ohne Probleme zu haben:

- Die Verwendung von AJAX erfordert zunächst die Verfügbarkeit von JavaScript auf dem Client.
- Die AJAX-Libraries müssen unterschiedliche Browser unterstützen (aktuell Microsoft Internet Explorer, Mozilla Firefox und Apple Safari).
- Die Verwendung der "Zurück"-Schaltfläche im Client kann zu unvorhergesehenen Problemen führen, da zwischenzeitlich eventuell Teile der Webseite geändert wurden.

42.5.2 Die AJAX-Controls

Die Microsoft-Variante von AJAX steht bereits seit Visual Studio 2008 zur Verfügung.

Folgende AJAX-Server-Controls können Sie nutzen:

- *ScriptManager*
- *ScriptManagerProxy*
- *UpdatePanel*
- *UpdateProgress*
- *Timer*

Im Folgenden wollen wir uns einige dieser Controls etwas näher ansehen.

[1] Kein Java, kein AJAX ...

ScriptManager

Dieses Control bildet die Grundlage für die Verwendung der anderen AJAX-Controls und muss **vor** diesen in der ASPX-Datei eingefügt werden.

Das *ScriptManager*-Control ermöglicht

- die Verwendung von Client-Skript-Funktionalität,
- das partielle Rendern von Webseiten auf dem Client (nur Teile der Seite werden geändert),
- den Zugriff auf Webdienste per Proxy-Klassen,
- die Verwendung von ASP.NET-Authentication in JavaScript-Klassen.

UpdatePanel

Zusammen mit dem *ScriptManager* können Sie mit diesem Control Seitenbereiche erstellen, die unabhängig vom Rest der Seite aktualisiert werden können (partielles Rendering). Dies bietet sich beispielsweise bei komplexen Seiteninhalten oder Detaildarstellungen an, in diesem Fall wird beispielsweise nur das *GridView* mit den Detaildaten aktualisiert

Das Handling ist ausgesprochen trivial: kopieren Sie einfach die Server-Controls in das *UpdatePanel,* die vom partiellen Seitenaufbau betroffen sein sollen.

HINWEIS: Sie können auf einer Seite auch mehrere *UpdatePanel*s erzeugen und diese erforderlichenfalls schachteln.

Werden jetzt von den Server-Controls im *UpdatePanel* Aktionen ausgelöst (Button-Klick, Navigationsschaltflächen etc.), wird nur der Inhalt des *UpdatePanel*s aktualisiert, ohne einen kompletten Seitenrefresh auszulösen.

BEISPIEL 42.40: Verwendung *UpdatePanel*

C#

Fügen Sie in eine neue Webseite ein *ScriptManager*-Control und ein *UpdatePanel* ein. Nachfolgend kopieren Sie bitte jeweils ein *Label*-Control und einen *Button* in den normalen Seitenbereich und in das *UpdatePanel.*

Die Ereignisroutinen:

```
protected void Page_Load(object sender, EventArgs e)
{
    Label1.Text = System.DateTime.Now.ToString();
    // Das Label im normalen Seitenbereich
}
```

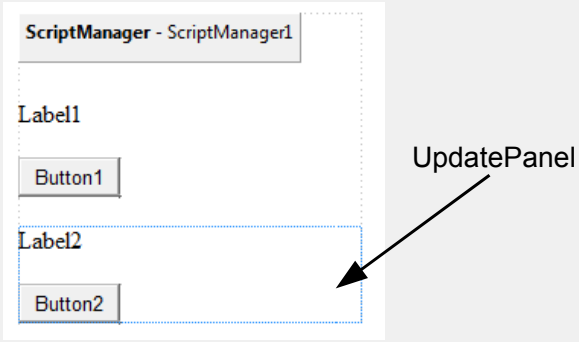
Hier arbeiten wir mit den Controls im *UpdatePanel:*

```
protected void Button2_Click(object sender, EventArgs e)
{
    Label2.Text = System.DateTime.Now.ToString();
```

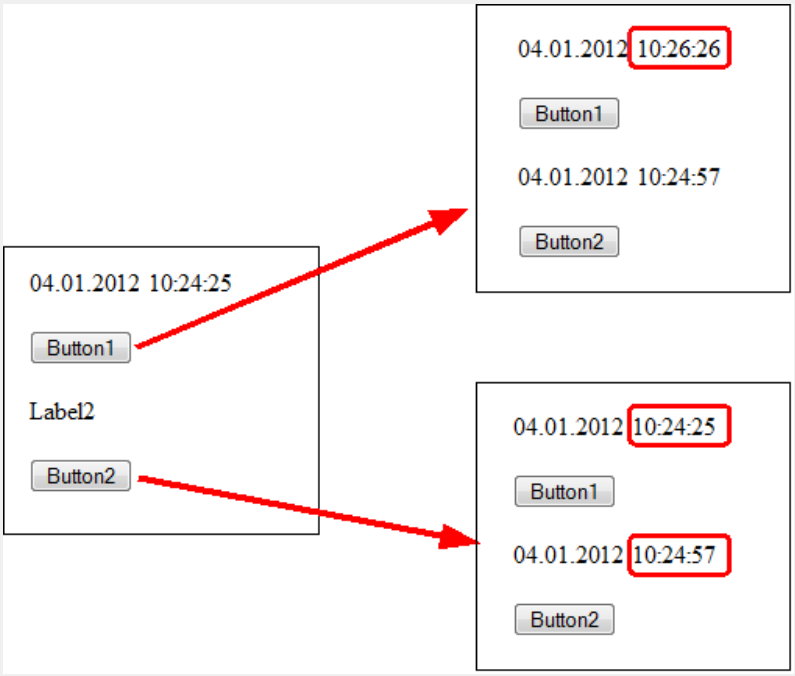
BEISPIEL 42.40: Verwendung *UpdatePanel*

```
}
```

Die Entwurfsansicht des Formulars:



Nach dem Start sollte in *Label1* bereits die aktuelle Zeit angezeigt werden (*Page_Load*). Klicken Sie jetzt auf auf *Button2*, ändert sich nur die Zeit in *Label2*, *Label1* bleibt unverändert, da kein Seitenabruf (*Page_Load*) erfolgt. Ein Klick auf *Button1* löst wiederum den normalen Seitenabruf aus, die Zeit in *Label1* ändert sich:



Bei der Verwendung des *UpdatePanel* gilt es allerdings auch einige Einschränkungen zu beachten.

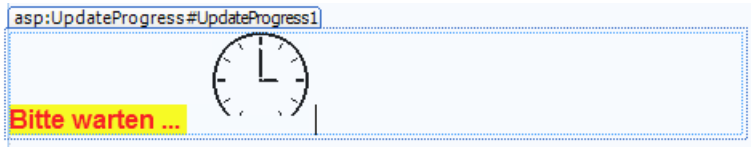
So bietet das Control **keine** Unterstützung für

- *TreeView* und *Menu*,
- WebParts,
- *FileUpload* mit asynchronem Postback,
- *GridView* und *DetailsView* mit aktiviertem *EnableSortingAndPagingCallbacks*,
- *Login, PasswordRecovery, ChangePassword* und *CreateUserWizard,* wenn diese nicht in editierbare Templates umgewandelt wurden,
- Substitution- und Validation-Controls.

UpdateProgress

Mit dem *UpdateProgress* steht Ihnen ein Control zur Verfügung, das auf dem Client immer dann eingeblendet wird, wenn ein *UpdatePanel* aktualisiert wird. Die Verzögerung für die Anzeige dieses Bereichs können Sie mit der *DisplayAfter*-Eigenschaft festlegen, um das Ausblenden brauchen Sie sich nicht zu kümmern, das erfolgt automatisch.

In das *UpdateProgress*-Control selbst fügen Sie beispielsweise Hinweistexte oder auch Grafiken (Animierte Gifs) ein, die dann während der Aktualisierung zu sehen sind:



Timer

Auf dieses Control hat sicher der eine oder andere schon lange gewartet. Endlich besteht die Möglichkeit, ohne eigene JavaScript-Programmierung aus dem Browser heraus zyklisch Ereignisse auf dem Server auszulösen. So können Sie beispielsweise Teile einer Webseite (bei Verwendung im *UpdatePanel)* oder auch die gesamte Website in bestimmten Intervallen aktualisieren.

HINWEIS: Der *Timer* initiiert ein Ereignis (*Tick)* auf dem Server!

BEISPIEL 42.41: *Timer* in einem *UpdatePanel* verwenden

C#

```
protected void Timer1_Tick(object sender, EventArgs e)
{
   Label3.Text = System.DateTime.Now.ToString();  //  Aktualisieren der Uhrzeit
}
```

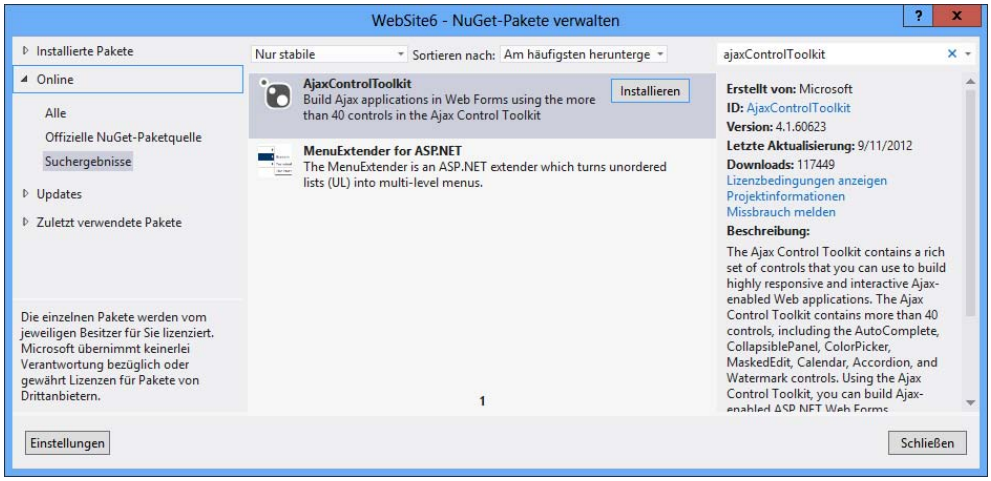
HINWEIS: Vorsicht ist beim Setzen der *Interval*-Eigenschaft angebracht. Zu kleine Werte führen zu hoher Netz- und Serverlast.

42.5.3 AJAX-Control-Toolkit

Neben den im vorhergehenden Abschnitt beschriebenen Controls bietet sich unter der Adresse

LINK: http://ajaxcontroltoolkit.codeplex.com/

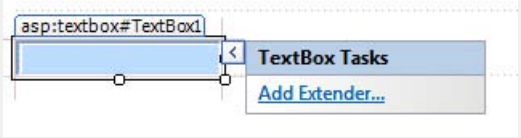
die Möglichkeit, das ASP.NET AJAX Control Toolkit herunterzuladen. Alternativ können Sie auch NuGet verwenden, so sparen Sie sich die Installation



Hierbei handelt es sich um eine Sammlung von eigenständigen Controls und Extendern. Letztere dienen dazu, vorhandene ASP.NET-Server-Controls, wie TextBox etc., um zusätzliche Funktionalität zu erweitern (per Client-Script).

BEISPIEL 42.42: Verwendung von Extendern

Die ASP.NET-TextBox verfügt nach Installation des AJAX-Control-Toolkits über ein neues Kontextmenü, mit dem Sie den gewünschten Extender hinzufügen können:



Eine Auswahl der möglichen Extender:

HINWEIS: Der Einsatz obiger Extender erfordert zwingend die Verwendung eines ScriptManagers auf der Webseite.

Nach dem Start der Seite verfügt die *TextBox* je nach Extender über erweitere Funktionalität, wie zum Beispiel eine Kalenderauswahl:

42.6 User Controls/Webbenutzersteuerelemente

Vollständigkeitshalber wollen wir in gebotener Kürze auch noch auf das Thema "Web-Benutzersteuerelemente" eingehen. Diese lassen sich in zwei grundsätzliche Gruppen einteilen:

- User Controls (zu deutsch Webbenutzersteuerelemente)
- Custom Controls (benutzerdefinierte Websteuerelemente)

User Controls

User Controls werden wie eine ASPX-Seite entworfen, d h., sie können neben dem obligaten Quellcode auch Steuerelemente enthalten (WYSIWYG-Entwurf). Abgespeichert werden User Controls in ASCX-Dateien. User Controls können Sie per Drag & Drop aus dem Projektmappen-Explorer in Ihr Anwendungsformular ziehen, eine Aufnahme in den Werkzeugkasten ist nicht möglich.

Custom Controls

Custom Controls müssen sich um ihr Äußeres per Quellcode kümmern, d.h., ein einfaches Zusammenklicken von fertigen Controls zu einem neuen Control ist nicht möglich. Derartige Controls können

- von *System.Web.UI.WebControls.WebControl* abgeleitet werden
- oder sie erben von vorhandenen Server Controls
- bzw. setzen sich aus mehreren Nachfahren der Klasse *System.Web.UI.Control* zusammen.

Diese Controls werden kompiliert und später als Assembly in die Webanwendungen eingebunden.

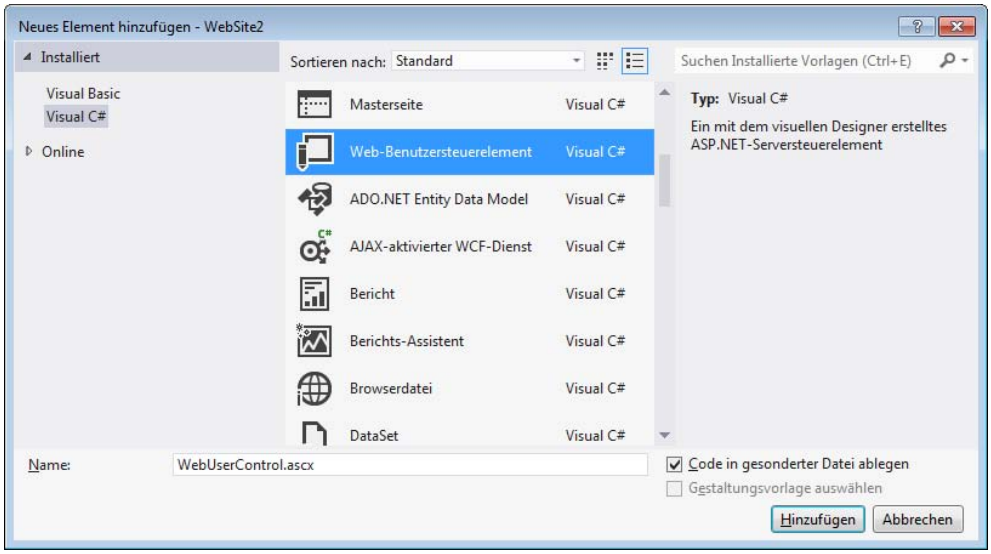
HINWEIS: Im Weiteren beschränken wir uns auf Beispiele für User Controls.

42.6.1 Ein simples Einstiegsbeispiel

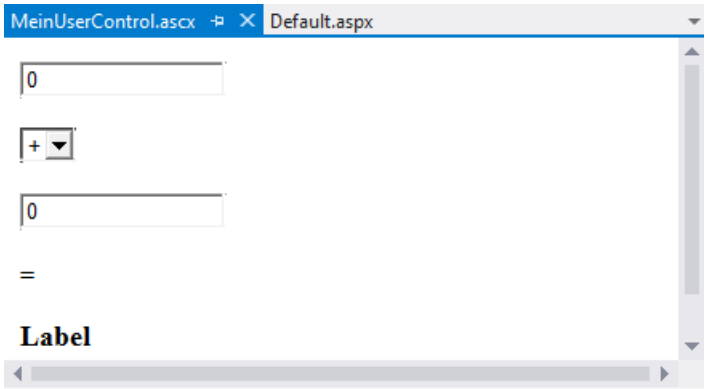
Ein ziemlich einfacher Rechner soll Ihnen den Einstieg in die Programmierung von User Controls demonstrieren.

Entwurf

Erzeugen Sie zunächst ein neues Web-Projekt und fügen Sie ein leeres Web-Control hinzu (Menüpunkt *Website|Neues Element hinzufügen|Web-Benutzersteuerelement*) hinzufügen:



Fügen Sie in das neu erstellte User Control zwei *TextBox*-Controls, eine *DropDownList* und ein *Label* ein:



Der *DropDownList* können Sie bereits jetzt über die Eigenschaft *Items* die Werte +, -, *, / zuweisen.

Quellcode

Die Berechnung (Methode *calc*) nehmen wir im User Control vor, der Aufruf dieser Routine bleibt jedoch dem übergeordneten Programm vorbehalten:

```
using System;
...
using System.Web.UI.HtmlControls;

public partial class MeinUserControl : System.Web.UI.UserControl
{
```

Die extern verfügbare Methode:

```
    public Single Calc()
    {
        Single a = Convert.ToSingle(TextBox1.Text);
        Single b = Convert.ToSingle(TextBox2.Text);
        Single c = 0;
```

Die Rechenoperation unterscheiden:

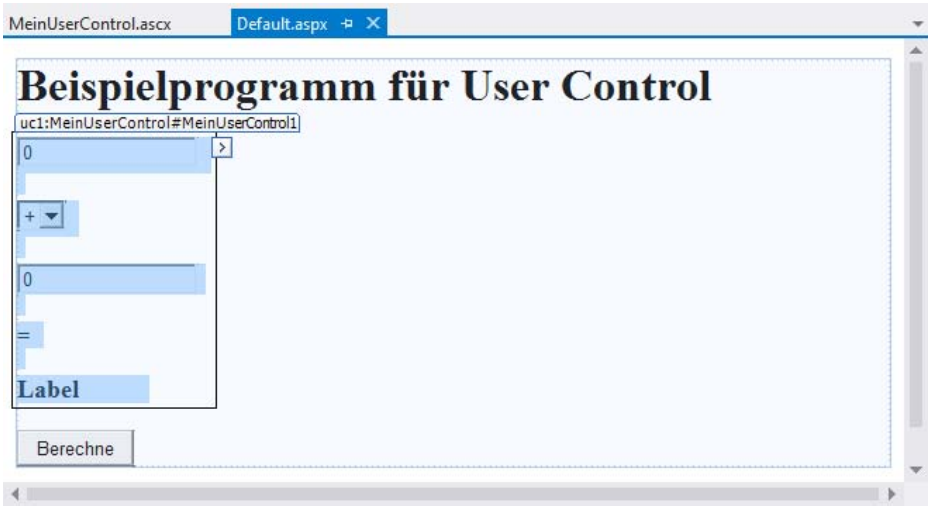
```
        switch (DropDownList1.SelectedValue)
        {
            case "+":
                c = a + b;
                break;
            case "-":
                c = a - b;
                break;
            case "*":
                c = a * b;
                break;
            case "/":
                c = a / b;
                break;
        }
        Label1.Text = c.ToString();
        return c;
    }
```

Beim Start initialisieren wir das Control:

```
    protected void Page_Load(object sender, EventArgs e)
    {
        Label1.Text = "";
    }
```

Einbindung

Wie schon beschrieben genügt es, wenn Sie das neu erstellte Control per Drag&Drop in das Web Form einfügen:



Wie Sie sehen, sind sofort alle enthaltenen Controls sichtbar, das Control ist bereits voll verwendbar!

Zum Aufruf der Methode *calc* nehmen wir einen *Button.*

Test

Starten Sie die Anwendung, fügen Sie sinnvolle Werte in die Textboxen ein und klicken Sie zum Abschluss auf den *Button*. Die Berechnung wird gestartet und das Ergebnis in *Label1* angezeigt.

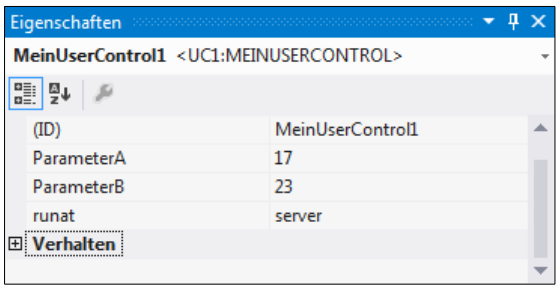
Eigene Eigenschaften definieren

Bisher gibt sich unser User Control noch recht "verschlossen", wir wollen deshalb zwei Eigenschaften hinzufügen, mit denen von "außen" auf die Textboxen zugegriffen werden kann. Erweitern Sie dazu die Klassendefinition des Controls um folgende Anweisungen:

```
...
public partial class MeinUserControl : System.Web.UI.UserControl
{
    Single _a = 0;
    Single _b = 0;

    public Single ParameterB
    {
        get { return _b; }
        set {
            _b = value;
```

```
            TextBox2.Text = _b.ToString();
            }
    }

    public Single ParameterA
    {
        get { return _a; }
        set {
             _a = value;
             TextBox1.Text = _a.ToString();
            }
    }
```

Nach dem Aktualisieren des User Controls sollten im Eigenschaften-Fenster auch die beiden neuen Eigenschaften auftauchen:



Fazit

Bereits mit wenig Aufwand können Sie Programmieraufgaben in User Controls kapseln und mit definierter Schnittstelle für Ihre Webanwendung bereitstellen. Zum einen vermeiden Sie aufgeblähte Anwendungen, in denen immer alles neu zu erfinden ist, zum anderen können Routineaufgaben von den User Controls sicher übernommen werden.

42.6.2 Dynamische Grafiken im User Control anzeigen

Vielleicht ist auch Ihnen beim Betrachten des Werkzeugkastens ein Manko aufgefallen: Es gibt kein Control, mit dem Sie Grafiken zur Laufzeit erzeugen können.

Es besteht zwar die Möglichkeit, mittels GDI+ eine Grafik (*jpeg/gif*) zu erzeugen, Sie müssen diese jedoch im Normalfall als Datei abspeichern und deren URL zum Beispiel für ein *Image*-Control nutzen. Dies hat zwei wesentliche Nachteile:

- Sie müssen für beliebige viele Nutzer Grafik-Dateien erzeugen, die Sie auch irgendwann wieder löschen müssen.
- Meist fehlen die nötigen Zugriffsrechte auf dem Server, um derartige Dateien zur Laufzeit zu erzeugen.

Dynamisches Senden der Grafik

Beide Probleme kann man vermeiden, wenn die Daten dynamisch via *Response*-Objekt an den Client gesendet werden.

BEISPIEL 42.43: Senden einer Grafik an den Client

C#
```
protected void Page_Load(object sender, EventArgs e)
{
   Response.Clear();
   Bitmap img = new Bitmap(200, 100);
   Graphics g = Graphics.FromImage(img);
   Rectangle rect = new Rectangle(0, 0, 200, 150);
   LinearGradientBrush myBrush  = new LinearGradientBrush(rect, Color.Red,
                                  Color.Yellow, LinearGradientMode.Vertical);
   g.FillRectangle(myBrush, new Rectangle(0, 0, 200, 100));
   ...
   Response.ContentType = "image/jpeg";
   img.Save(Response.OutputStream, ImageFormat.Jpeg);
   Response.End();
...
```

Das funktioniert soweit ganz gut, aber auf diesem Weg bekommen Sie nur die Grafik zum Client, die HTML-Seite bleibt auf der Strecke.

Abhilfe soll ein eigenes User Control schaffen, das wie ein Server Control in der Seite platziert werden kann und das zur Laufzeit einen grafischen Counter anzeigt.

Entwurf des Controls

Fügen Sie Ihrem Projekt ein *Web-Benutzersteuerelement* hinzu und legen Sie den Namen des Controls mit *SpezialCounter.ascx* fest.

HINWEIS: Auch wenn es vielleicht verlockend ist: Fügen Sie kein Control in das neue Steuerelement ein. Wir erzeugen allen nötigen Ausgaben per Code.

Wechseln Sie in die Datei *SpezialCounter.ascx.cs* und fügen Sie folgenden Code hinzu:

```
using System;
...
```

Einige Namespaces, die wir im Weiteren nutzen werden:

```
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;

public partial class SpezialCounter : System.Web.UI.UserControl
{
```

Wir deklarieren eine Eigenschaft *Anzahl*, die den anzuzeigenden Wert vorgibt:

```
private int myanzahl;

[Category("Allgemeines"), Browsable(true),
DefaultValue("0"),Description("Aktueller Zählerstand")]
public int Anzahl
{
    get { return myanzahl;}
    set { myanzahl = value;}
}
```

Wer jetzt vielleicht vermutet hat, dass wir *Form_Load* für die Grafikausgabe nutzen, liegt falsch. Sie könnten hier zwar eine Grafikausgabe realisieren, von Ihrer umgebenden HTML-Seite würde jedoch nichts übrig bleiben. Die Gründe: Das Ereignis tritt zu zeitig auf (noch vor dem Rendern der HTML-Seite) und es wird ausschließlich eine Grafik (*Response.End*) zurückgegeben.

Deshalb überschreiben wir die *Render*-Methode des Controls:

```
protected override void Render(HtmlTextWriter output)
{
```

Wir ermitteln den URL der umgebenden Seite

```
    string myurl = Request.Url.ToString();
```

und basteln uns daraus einen Querystring:

```
    myurl = String.Concat(myurl, "?", this.UniqueID, "=", Anzahl.ToString());
```

HINWEIS: Die Eigenschaft *this.UniqueID* gibt den eindeutigen Bezeichner des Controls zurück (*SpezialCounter1*, *SpezialCounter2* ...).

Die einzige Ausgabe unseres Controls bliebt zunächst die folgende Zeile:

```
    output.Write("<img id={0} src={1}>", this.UniqueID, myurl);
}
```

Was rendert die ASP.NET-Engine aus unserem Control? Ein kleiner Test und ein Blick in den Seiten-Quellcode sorgt für Aufschluss:

```
...
<img id=SpezialCounter1 src=http://localhost:1030/UserControl/default.aspx?SpezialCounter1=0>
...
```

Das Control erzeugt lediglich einen *img*-Tag, was zunächst nicht sonderlich interessant ist, wesentlich wichtiger ist die Quelle der Grafik: Es handelt sich um die aufrufende Seite mit einem zusätzlichen Querystring, den wir im Weiteren auswerten wollen.

Dazu nutzen wir die *OnInit*-Methode des neuen Controls, die wir überschreiben müssen:

```
protected override void OnInit(EventArgs e)
{
```

Stimmt der Querystring mit der aktuellen Controlinstanz überein?

```
        if (Request.Params[UniqueID] != null)
        {
```

Wenn ja, dann können wir hier den Counterwert aus dem Querystring bestimmen (Konvertieren von *String* in *Int* ist notwendig):

```
            int count = Convert.ToInt32(Request.Params[UniqueID]);
```

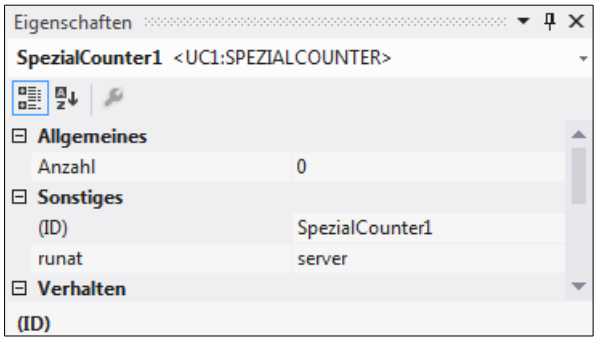
Jetzt können wir für die spezifische Anforderung des Clients (URL der Grafik) eine Grafik zurücksenden:

```
            Response.Clear();
            Bitmap img = new Bitmap(200, 80);
            Graphics g = Graphics.FromImage(img);
            Rectangle rect = new Rectangle(0, 0, 200, 80);
            LinearGradientBrush myBrush = new LinearGradientBrush(rect, Color.Navy,
                                          Color.White, LinearGradientMode.Vertical);
            g.FillRectangle(myBrush, new Rectangle(0, 0, 200, 80));
            myBrush = new LinearGradientBrush(rect, Color.Yellow, Color.Red,
                         LinearGradientMode.Vertical);
            SizeF size = g.MeasureString(count.ToString(), new Font("Arial", 48));
            g.DrawString(count.ToString(), new Font("Arial", 48), myBrush,
                        (200 - size.Width) / 2, 4);
            Response.ContentType = "image/jpeg";
            img.Save(Response.OutputStream, ImageFormat.Jpeg);
            g.Dispose();
            img.Dispose();
            Response.End();
        }
    }
}
```

Verwendung und Test

Wie kommt das neue Steuerelement in Ihre Web Form? Ganz einfach, per Drag&Drop aus dem Projektmappen-Explorer:

Unser *SpezialCounter* sieht zwar zu diesem Zeitpunkt noch recht hässlich aus, das wird sich aber zur Laufzeit ändern. Ein Blick in den Eigenschaften-Editor zeigt auch die von uns definierte Eigenschaft *Anzahl*:



Jetzt fügen wir dem Web Form noch etwas Code hinzu, um auch über mehrere Seitenaufrufe hinweg einen Counter zu realisieren

```
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
            Session["count"] = 1;
        else
        {
            SpezialCounter1.Anzahl = (int)Session["count"];
            Session["count"] = (int)Session["count"] + 1;
        }
    }

    protected void Button1_Click(object sender, EventArgs e)
    {
        SpezialCounter2.Anzahl = (int)Session["count"]+1000;
    }
}
```

Mit dem Start der Anwendung müssten beide Counter zunächst null anzeigen, nach einem Klick auf den Button wird *SpezialCounter1* über das *Form_Load*-Ereignis aktualisiert, *SpezialCounter2* über das Klick-Ereignis des Buttons:

42.6.3 Grafikausgaben per User Control realisieren

Wem das vorhergehenden Beispiel noch zu speziell ist, der kann es ja mal mit diesem User Control versuchen.

Ziel ist die Ausgabe einer beliebigen Bitmap im GIF-Format. Das Control soll sich um das Handling und das Speichern der Grafik kümmern, die Bitmap selbst wird zur Laufzeit durch den Nutzer erzeugt. Temporäre Dateien auf der Festplatte werden wir nicht benötigen, wie wir dies bewerkstelligen wurde bereits im vorhergehenden Abschnitt erläutert.

Das User Control

Auch in diesem Fall erzeugen wir zunächst ein Benutzersteuerelement und vergeben den Namen *Canvas.ascx*. Der Quellcode des Controls:

```
...
using System.ComponentModel;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;

public partial class Canvas : System.Web.UI.UserControl
{
    private Bitmap myBMP;
```

Unsere Schnittstelle nach außen (Eigenschaft *Bitmap*):

```
    public Bitmap Bitmap
    {
        get
          { return (Bitmap) Session[String.Concat(this.UniqueID, "_Bitmap")]; }
        set
          { Session[String.Concat(this.UniqueID, "_Bitmap")] = value; }
    }
```

HINWEIS: Wem die obigen Aufrufe "böhmisch" vorkommen: wir speichern die übergebene Bitmap in einer *Session*-Variablen und rufen Sie auch von dort wieder ab.

Ein *img*-Tag erzeugen und den Link für die Grafik generieren:

```
protected override void Render(HtmlTextWriter output)
{
    string myurl = Request.Url.ToString();
    myurl = String.Concat(myurl, "?", this.UniqueID, "=1");
    output.Write("<img id={0} src={1}>", this.UniqueID, myurl);
}
```

Fordert der Client die Grafik an,

```
protected override void OnInit(EventArgs e)
{
    if (Request.Params[UniqueID] != null)
    {
```

müssen wir zunächst prüfen, ob es schon eine Grafik gibt. Um unschöne Effekte zu vermeiden, liefern wir im Zweifelsfall eine Mini-Grafik (1x1 Pixel) zurück:

```
        if (Session[String.Concat(this.UniqueID, "_Bitmap")] != null)
            myBMP = (Bitmap)Session[String.Concat(this.UniqueID, "_Bitmap")];
        else
            myBMP = new Bitmap(1, 1);
        Response.Clear();
```

Konvertieren in GIF und Zurückschreiben an den Client:

```
        Response.ContentType = "image/Gif";
        myBMP.Save(Response.OutputStream, ImageFormat.Gif);
        Response.End();
    }
}
```

HINWEIS: Sie könnten statt der Bitmap auch die GIF-Grafik auf dem Server speichern (*Session*), diese ist kleiner als die Bitmap. Sie müssen dann aber auf eine mögliche Nachbearbeitung der Grafik verzichten.

Einbindung/Programmierung

Ziehen Sie einfach unser neues *Canvas*-Control aus dem Projektmappen-Explorer in die Arbeitsfläche. Mit Klick auf einen neuen Button werden wir die Bitmap erzeugen:

```
protected void Button2_Click(object sender, EventArgs e)
{
    int max = 400;
```

Zunächst ein Punktearray für die Ausgabe einer Sinuskurve:

```
    Single ampl = Convert.ToSingle(TextBox1.Text);
    PointF[] punkte = new PointF[max];
```

Dann die Bitmap und ein dazugehöriges *Graphics*-Objekt:

```
    Bitmap img = new Bitmap(400, 200);
    Graphics g = Graphics.FromImage(img);
    g.FillRectangle(new SolidBrush(Color.Yellow), new Rectangle(0, 0, 400, 200));
    Pen p = new Pen(Color.Black);
```

Wir berechnen die Koordinaten für die einzelnen Punkte:

```
    for (int i = 0; i < max; i++)
      punkte[i] = new PointF(i, Convert.ToSingle(ampl * Math.Sin(((Single)i)/10)) + 100);
```

Ausgabe der Kurve:

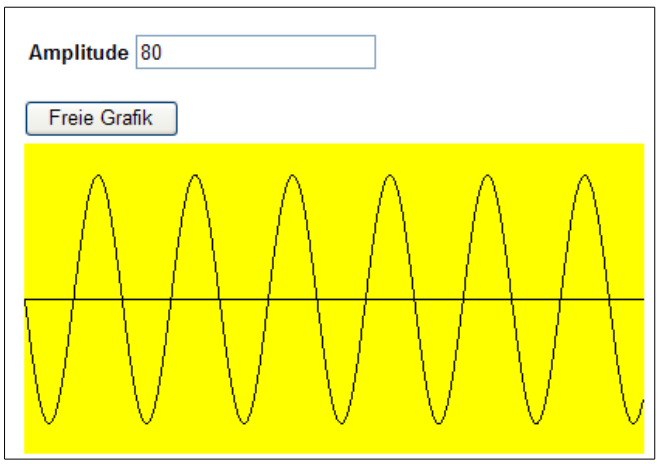
```
    g.DrawLines(p,punkte);
    g.DrawLine(p, 0, 100, 400, 100);
```

Zuweisen der *Bitmap* an das User Control:

```
    Canvas1.Bitmap = img;
}
```

Test

Starten Sie das Programm und klicken Sie auf den Button:



Mit etwas Phantasie fallen Ihnen sicher eine Menge von Anwendungsmöglichkeiten für Ihr neues User Control ein.

VIII

Teil VIII: Silverlight

- **Einführung in Silverlight**
- **Fenster und Seiten**
- **Datenbanken/Datenbindung**
- **Isolierter Speicher**
- **Fulltrust-Anwendungen**

Kapitel 43

Silverlight-Entwicklung

Neben den in den vorhergehenden Kapiteln ausgiebig vorgestellten WPF-Anwendungen für den Desktop bzw. für den Browser können Sie mit Visual Studio auch Silverlight-Applikationen entwerfen, die sich per Plug-in in eine Webseite einbinden lassen.

HINWEIS: Da sich WPF und Silverlight in weiten Teilen recht ähnlich sind, verweisen wir in Bezug auf Formularentwurf, Controls, Styling, Ereignisse, Datenbindung etc. auf die Kapitel 12 bis 16, um uns nicht zu wiederholen.

Im Mittelpunkt dieses Kapitels stehen die

- Silverlight-Grundlagen/Konzepte,
- die Unterschiede zu WPF,
- die Anbindung an die umgebende Website,
- Besonderheiten der Datenbindung,
- sowie einige spezielle Themen.

Doch jetzt wollen wir zunächst klären, worum es sich bei Silverlight eigentlich handelt, welche Zielplattformen es gibt und welche Unterschiede zu den WPF-Desktop-Anwendungen bestehen.

43.1 Einführung

Microsofts Silverlight ist der "kleine Bruder" des WPF-Frameworks für das Internet. Mit Silverlight können Sie, ähnlich wie in WPF, komplexe interaktive Webanwendungen (RIA = *Rich Internet Application*) auf der Basis Ihrer favorisierten .NET-Sprache schreiben und per Internet verteilen. Fast alles, was Sie in den vorhergehenden Kapiteln zum Thema "WPF" kennengelernt haben, lässt sich auch mit Silverlight realisieren. Das Besondere daran: Silverlight kommt mit einem "winzigen" Plug-in (Version 5 = 6.63 MB) auf dem Client aus, eine vorherige Installation des .NET-Frameworks auf dem Client ist nicht notwendig!

43.1.1 Zielplattformen

Da wir gerade von Plug-in reden, stellt sich sicher auch die Frage, auf welchen Zielplattformen (Browser, Betriebssysteme) Silverlight-Anwendungen zum Laufen gebracht werden können.

Das Silverlight-Plug-in in der derzeit aktuellen Version 5[1] (es handelt sich um ein klassisches ActiveX-Control) bietet derzeit Unterstützung an für

- den Internet Explorer (Windows, nur Desktop),
- Firefox (Windows),
- Opera (Windows, teilweise mit einigen Einschränkungen)
- und Safari (Mac-OS),
- sowie einige Unix-Derivate (Firefox).

Eine "kleine" Einschränkung müssen wir allerdings bereits an dieser Stelle treffen:

HINWEIS: Silverlight ist nicht im Internet-Explorer 10 für den Windows 8-Desktop lauffähig!

Genauere Informationen zu den unterstützten Windows- und Mac-Versionen finden Sie unter folgender Adresse:

LINK: `http://www microsoft.com/getsilverlight/Get-Started/Install/Default.aspx`

Informationen zu den unterstützten Unix-Derivaten bietet Novell (Projekt "Moonlight") unter folgender Adresse:

LINK: `http://mono-project.com/MoonlightSupportedPlatforms`

HINWEIS: Grundsätzlich sollten Sie ab und zu einen Blick auf obige Adressen werfen, hier sind immer wieder Änderungen zu erwarten.

43.1.2 Silverlight-Applikationstypen

In Visual Studio werden Ihnen derzeit vier Templates im Zusammenhang mit Silverlight angeboten:

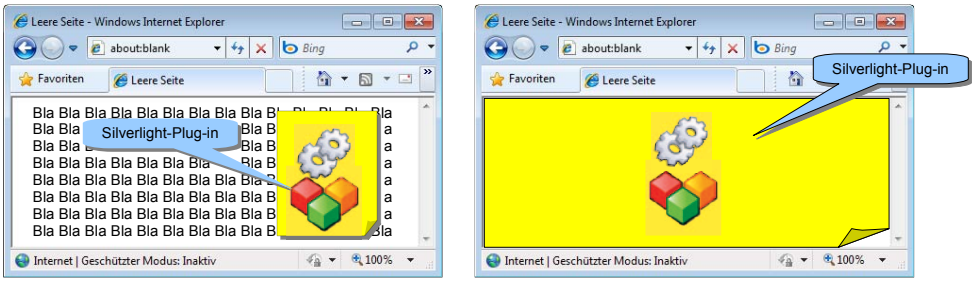
- Silverlight-Anwendung
- Silverlight-Navigationsanwendung
- Silverlight-Klassenbibliothek
- Silverlight-Geschäftsanwendung/WCF RIA Services-Klassenbibliothek

Im Folgenden wollen wir uns kurz mit den Unterschieden der beiden erstgenannten Anwendungstypen auseinandersetzen, um Ihnen die Entscheidung zu erleichtern.

[1] Die vermutlich letzte Version für einige Zeit ...

Silverlight-Anwendung

Grundsätzlich sollten Sie zunächst wissen, wofür und in welchem Rahmen Sie Ihre Silverlight-Anwendung einsetzen wollen. Zwei grundsätzliche Szenarien sind denkbar:

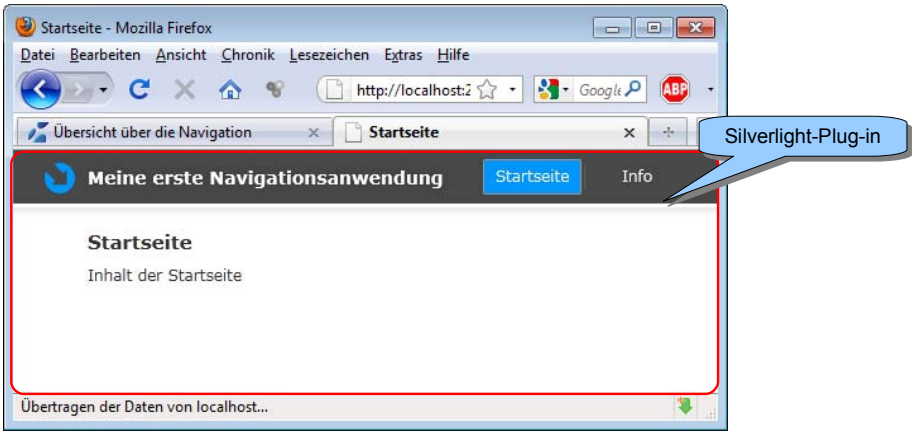


- Die linke Abbildung zeigt das Silverlight-Plug-in als Teil einer Website, d h., der Hauptteil der angezeigten Seite ist zunächst ein HTML-Dokument, in das ein oder auch mehrere Silverlight-Element(e) eingebettet wurden. Dies dürfte auch der Standardfall für die Verwendung einer **Silverlight-Anwendung** sein, meist beschränkt sich die Anwendung auf eine Seite.
- Die rechte Abbildung zeigt eine Silverlight-Anwendung als ausschließlichen Inhalt der Seite, d h., es werden keine HTML-Inhalte dargestellt. Soll das Plug-in mehrere Seiten darstellen, müssen Sie sich auch um die Navigation zwischen den Seiten kümmern, hier ist meist eine **Silverlight-Navigationsanwendung** die bessere Wahl.

Welche Dateien zu einer Silverlight-Anwendung gehören und welche Bedeutung sie haben, klären wir in den folgenden Abschnitten dieses Kapitels.

Silverlight-Navigationsanwendung

Dieser Spezialfall einer Silverlight-Anwendung ermöglicht durch ein vorgegebenes Grundgerüst die relativ einfache Navigation zwischen einzelnen Seiten einer Anwendung:



Mit Hilfe eines übergeordneten Frames, der quasi als Container fungiert, können Sie einzelne Seiten einblenden bzw. zwischen diesen navigieren. Dies kann programmgesteuert oder per Nutzeraktion erfolgen. Standardmäßig bleibt die Kopfzeile mit den Navigationsschaltflächen erhalten, lediglich die darunter befindlichen Inhalte wechseln. Sie haben jedoch die Möglichkeit, den Rahmen an Ihre Bedürfnisse anzupassen.

HINWEIS: Mehr zu diesem Anwendungstyp finden Sie im Abschnitt 43.4.5auf Seite 2133.

Last but not least, können Sie die beiden oben vorgestellten Anwendungstypen auch als Out-of-Browser-Anwendung betreiben.

Silverlight Out-of-Browser-Anwendung

Mit der Version 3 wurde Silverlight ein neues Konzept hinzugefügt: die Out-of-Browser-Applikation. Diese unterscheidet sich zunächst nicht weiter von den normalen Silverlight-Anwendungen, die *AppManifest*-Datei verfügt jedoch über entsprechende Einträge, die ein Installieren der Anwendung auf dem Client-PC ermöglichen. Dazu werden die Anwendungsdaten lokal kopiert und über eine entsprechende Verknüpfung mit der Anwendung *sllauncher.exe* ausgeführt. Nachfolgend ist die Anwendung gänzlich von Ihrer Website abgekoppelt. Werden keine Online-Daten abgerufen, ist ein reiner Offline-Betrieb möglich.

Mit den Details und der Vorgehensweise beschäftigt sich das Praxisbeispiel 43.8.1 "Eine Out-of-Browser-Applikation realisieren". Die mit diesem Anwendungstyp einhergehenden Problem werden in den Praxisbeispielen

- 43.8.2 Out-of-Browser-Anwendung aktualisieren
- 43.8.3 Testen auf aktive Internetverbindung
- 43.8.4 Auf Out-of-Browser-Anwendung testen

näher beleuchtet und natürlich auch gelöst.

43.1.3 Wichtige Unterschiede zu den WPF-Anwendungen

Wie schon mehrfach angemerkt, bestehen zum "großen Bruder" WPF einige Unterschiede, die es dem Entwickler nicht immer leicht machen, Projekte zu portieren. Dabei sind zwei Themenbereiche zu unterscheiden:

- Einschränkungen durch das Silverlight-Konzept
- Unterschiede durch unterschiedliche Codebasis

Einschränkungen durch das Silverlight-Konzept

Grundsätzlich sollten Sie zunächst einmal vom Gesamtkonzept "Silverlight als Plug-In des Web-Browsers" ausgehen. Bereits diese Tatsache bringt einige wesentliche Unterschiede mit sich, läuft doch Ihre Silverlight-Anwendung in einer so genannten Sandbox. Die damit einhergehenden Ein-

schränkungen beim Zugriff auf das Clientsystem (kein Zugriff auf Geräte, Laufwerke, Prozesse, die Tastatur im Vollbildmodus[1] etc.) sind konzeptbedingt und müssen so hingenommen werden.

Allerdings wurden diese Einschränkungen mit jeder weiteren Version etwas aufgeweicht, gerade seit der Version 4 haben Sie durch den neuen Full-Trust-Modus auch Zugriff auf

- das lokale Dateisystem,
- die lokale Hardware,
- Cross-Domain-Netzwerkzugriffe,
- COM-Automation-Objekte,
- Drag & Drop (siehe 43.8.10, "Drag & Drop realisieren"),
- Zwischenablage (siehe 43.8.11, "Auf die Zwischenablage zugreifen"),
- eigenständige Fenster (siehe 43.8.12, "Weitere Fenster öffnen") und
- die Tastatur im Vollbildmodus.

Allerdings ist in diesen Fällen die einmalig explizite Zustimmung des Endnutzers für den Wechsel in den Full-Trust-Modus erforderlich.

Unterschiede durch unterschiedliche Codebasis

Wie schnell ersichtlich sein wird, kann nicht der gesamte Funktionsumfang von WPF in Silverlight zur Verfügung stehen, dazu ist die nötige Codebasis einfach zu umfangreich. Allerdings dürfen Sie jetzt nicht den Schluss ziehen, dass es sich bei Silverlight um eine Teilmenge von WPF handelt. Vielmehr verfügen Silverlight und WPF über eine gemeinsame Schnittmenge an Funktionen, die Silverlight um eigenständige Funktionen erweitert. Etwas weniger verständlich sind allerdings kleinere Syntax-Unterschiede, die Sie recht schnell in Verzweiflung stürzen werden.

Ansonsten werden Sie mit einer sich pro Versionsnummer ständig ändernden Liste von marginalen Unterschieden konfrontiert.

Einige Auszüge in Bezug auf die aktuelle Version 5:

- keine dynamischen Ressourcen
- nur *BitmapImage* für *ImageSource* zulässig
- diverse Einschränkungen beim Datenzugriff (kein *DataSet,* XPath, *ObjectDataProvider* ...)
- Einschränkungen bei Triggern
- fehlende Controls (*Menu*, *MenuItem* ...)

An dieser Stelle wollen wir die Liste nicht zu lang werden lassen (wir wollen Ihnen ja nicht den Spaß zu früh verderben).

[1] Das soll ein Schutz vor Phishing-Attacken sein, bei denen Anmeldebildschirme nachgebildet werden.

43.1.4 Vor- und Nachteile von Silverlight-Anwendungen

Selbst auf die Gefahr hin, sich dabei "in die Nesseln zu setzen", wollen wir dieses Thema im Schnelldurchlauf diskutieren.

Zunächst einmal sollte man die Frage stellen, in Bezug auf welchen Anwendungstyp die Vor- bzw. Nachteile diskutiert werden sollen. Grundsätzlich kann natürlich kein Vergleich von Silverlight mit einer reinen Desktopanwendung erfolgen, diese hat ein ganz anderes Aufgabengebiet. Maximal könnten hier noch **WPF-ClickOnce**-Anwendungen in Betracht kommen, die mit ähnlich geringem Aufwand verteilt werden können. Möchten Sie mit Webanwendungen arbeiten, stehen im Wesentlichen die anderen beiden wichtigen Webtechnologien **ASP.NET/AJAX** bzw. **Adobe Flash** optional zur Auswahl.

WPF-ClickOnce-Anwendungen

Grundsätzlich handelt es sich bei WPF-ClickOnce-Anwendungen um reine Desktop-Anwendungen, die lediglich für eine schnelle Verteilung über das Internet optimiert sind. Eine Einbindung in Webseiten ist nicht vorgesehen, die Anwendung wird vom Desktop gestartet.

Pro:

- wesentlich größerer Funktionsumfang (siehe vorhergehender Abschnitt)
- keine Unterschiede in der Codebasis zu den "normalen" Anwendungen
- Erscheinung als eigenständige Anwendung ohne Browser (auch bei Silverlight im Rahmen einer Out-of-Browser-Anwendung möglich)
- komplette Interaktion mit Systemressourcen möglich
- mehrere Fenster gleichzeitig

Kontra:

- keine Integration in Webseiten
- wesentlich größerer Download (.NET-Framework bzw. Client Profil)
- Einschränkung bei der Anzahl der Gastsysteme ("nur" Windows-Plattform)

ASP.NET/AJAX

Mit ASP.NET/AJAX handelt es sich zunächst um eine reine Form der Webanwendung, die meist problemlos (Java-Zugriff vorausgesetzt) auf fast allen Gastsystemen lauffähig ist.

Pro:

- Geringe clientseitige Systemanforderungen
- kein Plug-In erforderlich

Kontra:

- teilweise grauenhafte Frickelei, um Daten in akzeptabler Zeit zwischen Client und Server abzugleichen
- wenig komfortable Controls
- Performance-Probleme (interpretiertes JavaScript)
- "grauenhafter" Entwurf für den bisherigen Desktop-Entwickler
- teilweise Browser-Kompatibilitätsprobleme

FLASH

Damit sind wir beim direkten Pendant von Silverlight angekommen. Adobe Flash (seit 1997 am Markt) dürfte wohl das Vorbild und der Antrieb gewesen sein, Silverlight zu entwickeln. Auch hier handelt es sich um ein Plug-In für diverse Browser, das den eigentlichen Code interpretiert und ausführt.

Ein Vergleich an dieser Stelle dürfte schnell zu Glaubensfragen führen, allerdings kann man in einem C#-Buch wohl auch davon ausgehen, dass Sie als Leser an einer Programmierung in einer .NET-Sprache, d h. in C#, interessiert sind.

Pro:

- fast vollständige Verbreitung des Plug-ins, damit wesentlich höhere Akzeptanz
- große Entwicklergemeinde
- keine Trennung Code und Design
- bisher teilweise noch Entwicklungsvorsprung (ab Version 4 nicht mehr relevant)

Kontra:

- keine Programmierung in .NET-Sprache
- keine Verwendung Ihrer bisherigen Libraries
- andere Codebasis als auf der Serverseite

Fazit

Wenn Sie bereits mit C# gearbeitet haben und auf Visual Studio als Entwicklungsumgebung setzen, dürfte die Wahl zwischen Flash und Silverlight sicher zu Gunsten von Silverlight ausfallen.

HINWEIS: Beachten Sie allerdings, dass die Installation des Plug-ins für viele Endanwender aus psychologischen und Sicherheits-Gründen zu einem Problem werden kann. Sie schränken bei Verwendung von Silverlight möglicherweise die Verbreitung ihrer Anwendung ein.

Sind die letztgenannten Fälle ein Problem für Sie, sollten Sie in den "sauren Apfel beißen" und sich für ASP.NET/Ajax entscheiden. In allen anderen Szenarien für reine Webanwendungen dürften Sie mit Silverlight am besten bedient sein.

43.1.5 Entwicklungstools

An dieser Stelle beschränken wir uns auf einen Verweis zum Abschnitt 14.9.1, neben Visual Studio können Sie auch *Microsoft Expression Blend* für den Entwurf einsetzen. Auch für den Entwurf von Silverlight-Anwendungen gilt: Visual Studio eignet sich besser für den Programmierer, Blend bietet hingegen dem Designer mehr Möglichkeiten.

43.1.6 Installation auf dem Client

Voraussetzung für die Lauffähigkeit Ihrer Silverlight-Anwendung auf dem Client ist die Installation eines entsprechenden Plug-in auf dem Client-PC. Und damit kommen wir zum einfachsten und gleichzeitig auch schwierigsten Thema des Kapitels. Prinzipiell ist die Installation des Silverlight-Plug-in trivial:

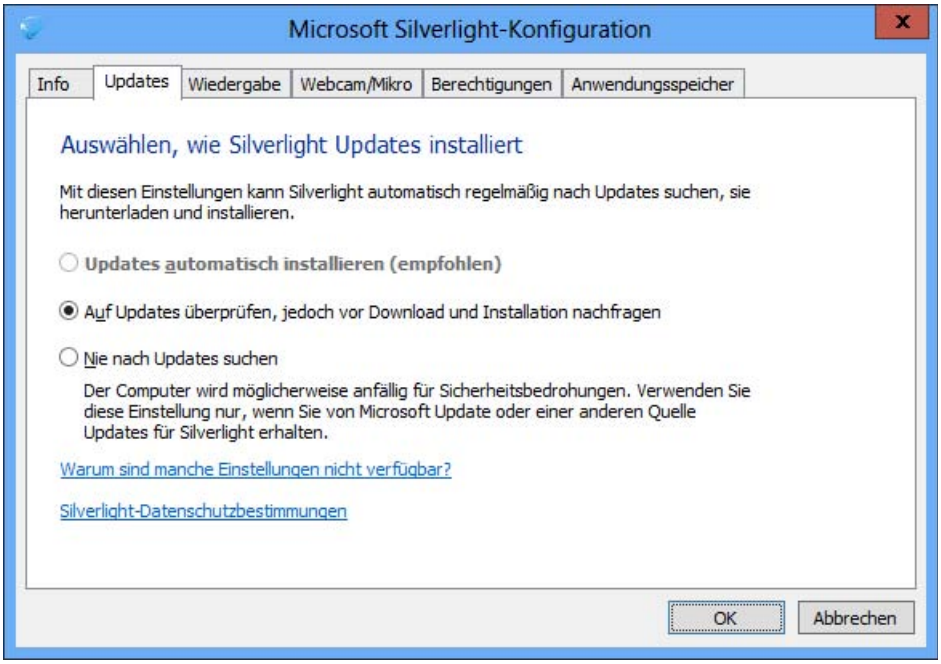
- Der Browser meldet bei einer Website, die das Silverlight-Plug-in enthält, dass zunächst Microsoft Silverlight installiert werden muss:

- Nach dem Klick auf dieses Symbol unterscheidet sich der weitere Verlauf je nach Browser etwas: In Firefox müssen Sie die Datei *Silverlight.exe* zunächst herunterladen und dann explizit ausführen, im Internet Explorer können Sie gleich die Option "Ausführen" wählen.
- In jedem Fall wird der Anwender mit einer Sicherheitsmeldung konfrontiert, die ihn auch drauf hinweist, welche Gefahren durch die Ausführung entstehen können:

  Arbeiten Sie mit Windows Vista/7/8 wird noch eine entsprechende Sicherheitsberechtigung (UAC/Benutzerkontensteuerung) abgefragt. Unsichere Anwender werden wohl an dieser Stelle abbrechen und damit nicht in den "Genuss" Ihrer Silverlight-Anwendung kommen.
- Stimmt der Anwender der Option "Ausführen" zu, wird nach der Anzeige von Lizenzvertrag und Datenschutzbestimmungen Silverlight installiert.

Updates/Deinstallation

Silverlight prüft standardmäßig auf neue Updates und installiert diese auch selbst. Die entsprechenden Einstellungen finden Sie, wenn Sie mit der rechten Maustaste auf das laufende Plug-in klicken:



Belassen Sie es im Normalfall bei dieser Standardeinstellung, gerade auch wegen der recht kurzen Versionswechsel/Updates bei Silverlight.

HINWEIS: Möchten Sie eine komplette Deinstallation vornehmen, müssen Sie dies über die Systemsteuerung (Programme) erledigen.

43.2 Die Silverlight-Anwendung im Detail

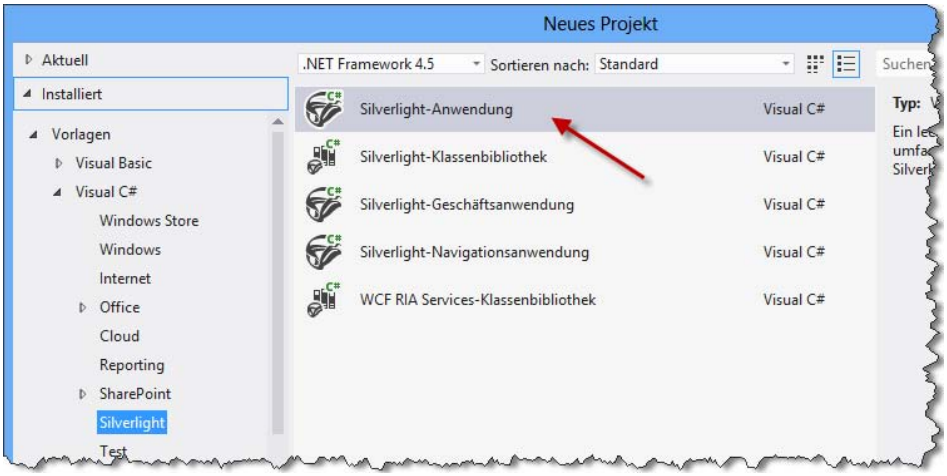
Nachdem wir Sie jetzt mit allgemeinen Ausführungen rund um das Thema "Silverlight" gelangweilt haben, möchten wir zu den interessanteren Themen übergehen.

Zunächst werden wir uns anhand eines trivialen Einstiegsbeispiels den grundsätzlichen Aufbau einer Silverlight-Anwendung näher ansehen. Dabei sind für uns vor allem die zum Projekt gehörenden Dateien und deren Bedeutung von Interesse.

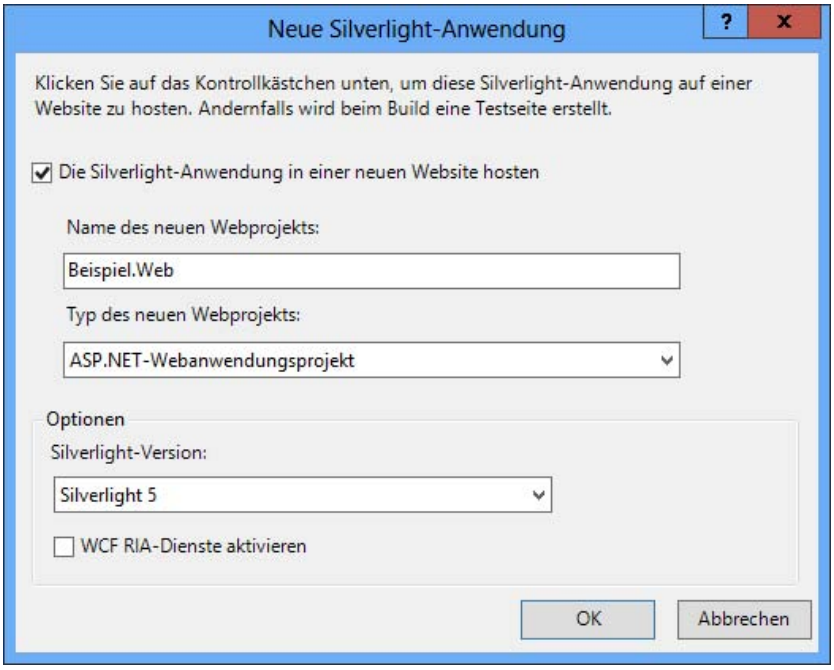
43.2.1 Ein kleines Beispielprojekt

Hier die wichtigsten Schritte:

- Wählen Sie zunächst aus den drei Silverlight-Vorlagen die "Silverlight-Anwendung" aus:



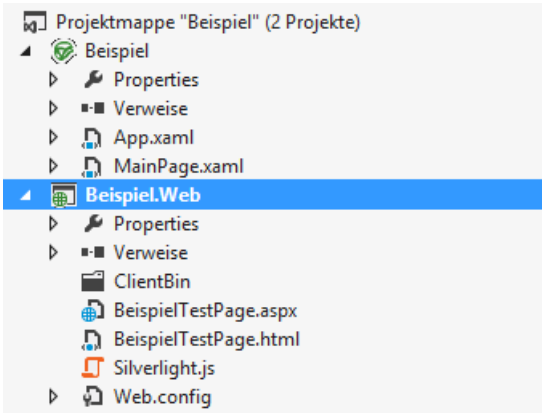
- Nachdem Sie einen Projektnamen festgelegt und den *OK*-Button betätigt haben, erscheint ein neuer Assistent. In diesem müssen Sie entscheiden, ob für Ihre Silverlight-Anwendung ein Web-Projekt erstellt werden soll oder nur eine einfache Testseite.



Dies ist nötig, da Silverlight zunächst immer im Rahmen einer Webseite dargestellt und ausgeführt wird. Wir empfehlen Ihnen hier die Verwendung eines "ASP.NET-Webanwendungsprojekt" als Webprojekt, da Sie in diesem Fall bereits eine recht brauchbare Zusammenstellung der nötigen Installationsdateien erhalten. Doch dazu später mehr.

Zusätzlich können Sie sich hier auch für die gewünschte Silverlight-Version entscheiden (es muss nicht immer die neueste Version sein) und optional die Unterstützung für RIA-Dienste aktivieren (mehr dazu später).

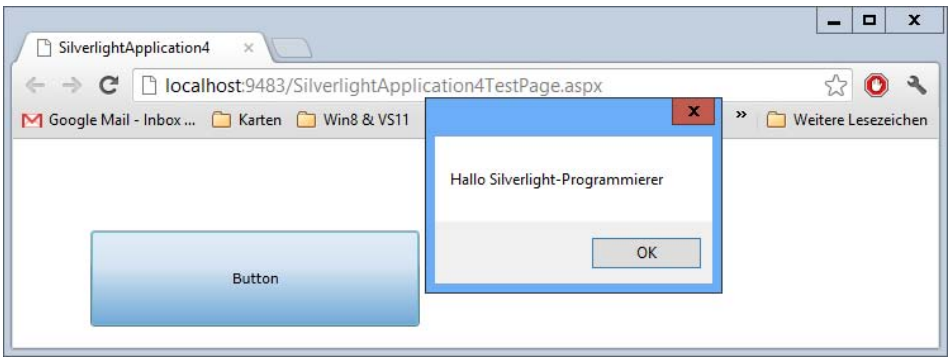
- Ihr noch "jungfräuliches" Projekt sollte jetzt aus folgenden Dateien bestehen, auf die wir in den weiteren Abschnitten im Detail eingehen wollen:



- Fügen Sie der Seite *MainPage.xaml* eine Schaltfläche (Button) hinzu und erstellen Sie einen dazugehörigen Ereignishandler mit folgendem Inhalt:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo Silverlight-Programmierer");
}
```

- Starten Sie jetzt die Anwendung und überzeugen Sie sich von der Funktionsweise:



Nachfolgend wollen wir uns ansehen, was für Dateien zu diesem Testprojekt gehören und welche Funktion sie haben.

43.2.2 Das Application Package und das Test-Web

Bevor Sie jetzt im falschen Verzeichnis landen und die falschen Dateien begutachten, ein Hinweis:

HINWEIS: Die final erzeugten Dateien finden Sie im Anwendungsordner des Test-Webs, nicht im *bin\Debug*-Verzeichnis der Silverlight-Anwendung.

Folgende Dateien sollten Sie vorfinden:

- *BeispielTestPage.aspx* und *BeispielTestPage.html*
- *Silverlight.js*
- *web.config*
- im Unterverzeichnis *\ClientBin* die Datei *Beispiel.xap*

Sie ahnen es sicher schon, Ihre Silverlight-Anwendung steckt in der Datei *Beispiel.xap*, um die wir uns als zunächst kümmern wollen.

.xap-Package

Bei dieser Datei (dem Application Package) handelt es sich um ein ZIP-Paket, das alle nötigen Assemblies und Ressourcen ihrer Silverlight-Anwendung enthält (keine Videos).

In jedem Fall sind mindestens zwei Dateien in diesem Package enthalten:

- *AppManifest.xaml*
- sowie die kompiliert *.dll* mit dem Eintrittspunkt Ihrer Anwendung.

Die Datei *AppManifest.xaml* enthält zunächst nur den Namen der Anwendungsassembly, kann aber auch Informationen zur Distribution (Out-of-Browser-Anwendung) enthalten. Bitte bearbeiten Sie diese Datei nicht direkt!

Silverlight.js

Hierbei handelt es sich um eine JavaScript-Datei aus der Silverlight SDK die eine Reihe von Funktionen für die Initialisierung des Silverlight-Plug-in bereitstellt.

Da wir uns nicht mit dem Erstellen des Silverlight-Plug-in per JavaScript beschäftigen wollen, gehen wir auf die Funktionen in dieser Datei nicht weiter ein.

HINWEIS: Die Datei wird in den beiden Testseiten (siehe folgender Abschnitt) referenziert.

...TestPage.aspx & ...TestPage.html

Hierbei handelt es sich um zwei einfache Webseiten, die lediglich die Aufgabe haben, das Silverlight-Plug-in aufzunehmen. Sie können den betreffenden Codeabschnitt auch in eigene Webseiten kopieren.

BEISPIEL 43.1: Grundaufbau von ...*TestPage.aspx*

HTML

```
#<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >

<head>
...
```

Die Utility-Datei *Silverlight.js* referenzieren:

```
    <script type="text/javascript" src="Silverlight.js"></script>
```

Hier folgt eine einfache Fehlerbehandlung, falls das Plug-in nicht korrekt initialisiert werden kann, oder wenn ein nicht behandelter Fehler in der Silverlight-Anwendung auftritt. Der Fehler (bzw. dessen Ursache) wird als einfache Meldung der Webseite ausgegeben:

```
    <script type="text/javascript">
        function onSilverlightError(sender, args) {
            var appSource = "";
            if (sender != null && sender != 0) {
              appSource = sender.getHost().Source;
            }
            var errorType = args.ErrorType;
            var iErrorCode = args.ErrorCode;
            if (errorType == "ImageError" || errorType == "MediaError") {
              return;
            }
            var errMsg = "Unbehandelter Fehler in Silverlight-Anwendung" +
                         appSource + "\n" ;
            errMsg += "Code: "+ iErrorCode + "    \n";
            errMsg += "Kategorie: " + errorType + "       \n";
            errMsg += "Meldung: " + args.ErrorMessage + "     \n";
            if (errorType == "ParserError") {
                errMsg += "Datei: " + args.xamlFile + "     \n";
                errMsg += "Zeile: " + args.lineNumber + "     \n";
                errMsg += "Position: " + args.charPosition + "     \n";
            }
            else if (errorType == "RuntimeError") {
                if (args.lineNumber != 0) {
                    errMsg += "Zeile: " + args.lineNumber + "     \n";
                    errMsg += "Position: " +  args.charPosition + "     \n";
                }
                errMsg += "Methodenname: " + args.methodName + "     \n";
            }
```

BEISPIEL 43.1: Grundaufbau von ...*TestPage.aspx*

```
            throw new Error(errMsg);
        }
    </script>
</head>
<body>
```

Hier erfolgt die eigentliche Einbindung des Silverlight-Plug-in:

```
    <form id="form1" runat="server" style="height:100%">
    <div id="silverlightControlHost">
```

Beachten Sie, dass auch eine Reihe von Parametern übergeben wird, auf die wir in der nachfolgenden Tabelle noch kurz eingehen werden:

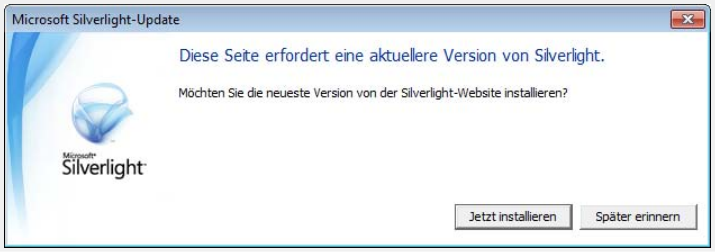
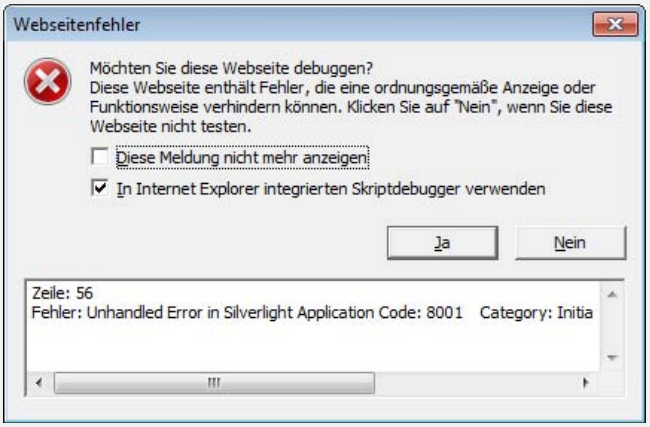
```
        <object data="data:application/x-silverlight-2," type="application/x-silverlight-2"
                width="100%" height="100%">
          <param name="source" value="ClientBin/Beispiel.xap"/>
          <param name="onError" value="onSilverlightError" />
          <param name="background" value="white" />
          <param name="minRuntimeVersion" value="5.0.61118.0" />
          <param name="autoUpgrade" value="true" />
```

Hier folgen alternative Links, falls das Plug-in nicht auf dem Client-PC installiert ist:

```
         <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=5.0.61118.0"
             style="text-decoration:none">
       <img src="http://go.microsoft.com/fwlink/?LinkId=161376"
             alt="Holen Sie sich Microsoft Silverlight" style="border-style:none"/>
     </a>
     </object><iframe id="_sl_historyFrame"
         style="visibility:hidden;height:0px;width:0px;border:0px"></iframe></div>
    </form>
</body>
</html>
```

Welche Bedeutung die einzelnen Parameter beim Initialisieren des Plug-in haben zeigt die folgende Tabelle:

Parameter	Beschreibung
source	Dieser Parameter spezifiziert den URI Ihrer Silverlight-Anwendung, d h., des Application Packages (*.xap*-Datei). Die *.xap*-Datei kann sich auf dem Host-Server oder auf einem Remote-Server befinden. (Möglichkeit des Load-Balancing).
width, height	Hiermit weisen Sie die Höhe und Breite des Silverlight-Plug-in zu. Sie können einen Wert in Pixeln oder in Prozenten angeben.
background	Mit *background* bestimmen Sie die Farbe des Silverlight-Plug-in-Hintergrunds. Sie können einen hexadezimalen Farbwert oder den Namen der Farbe angeben.

Parameter	Beschreibung
minRuntimeVersion	Hierbei handelt es sich um die niedrigste Runtime-Version, die von Ihrer Silverlight-Applikation gefordert wird. Ist diese nicht installiert, wird dem Anwender eine entsprechende Meldung angezeigt: 
autoUpgrade	Ist auf dem Client-PC Silverlight installiert, werden automatisch auch zukünftige Versionen installiert, wenn der Anwender dieses Feature nicht im Silverlight-Konfigurationsdialog (rechte Maustaste, Eigenschaften) deaktiviert. Alternativ können Sie die Art der Aktualisierung mit diesem Parameter vorgeben.
onError	Dieses Ereignis wird dann ausgelöst, wenn eine Ausnahme in Ihrer Anwendung nicht behandelt wurde. Die entsprechende JavaScript-Funktion zur Anzeige dieser Meldung ist bereits in die Testseite integriert, sollte aber aus optischen Gründen angepasst werden: 

Damit haben wir zunächst alle Dateien betrachtet, die Sie nach einem erfolgreichen Compilerlauf für die Distribution benötigen. Wir können uns nun den "Innereien" der Silverlight-Anwendung zuwenden.

43.3 Die Projektdateien im Überblick

Nachdem der erste Test lediglich ein nettes kleines Application Package hervorgebracht hat, müssen wir uns nun wohl oder übel mit den Projektdateien beschäftigen, d h. mit Ihrem "Arbeitsplatz" als Entwickler.

43.3.1 Projektverwaltung mit App.xaml & App.xaml.cs

Der Eintrittspunkt in Ihre Silverlight-Anwendung wird durch die Datei *App.xaml* bzw. die dazugehörige Quellcode-Datei *App.xaml.cs* verwaltet.

In der *App.xaml*-Datei können Sie vor allem anwendungsweite Ressourcen unterbringen. Viel interessanter ist aber die Datei *App.xaml.cs*, in der die *App*-Klasse mit einigen wichtigen Ereignissen definiert wird.

BEISPIEL 43.2: Beispiel für App.xaml & App.xaml.cs

XAML

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="SilverlightApplication4.App" >
    <Application.Resources>

    </Application.Resources>
</Application>
```

C#

```
namespace Beispiel
{
    public partial class App : Application
    {
```

Zuweisen der Ereignishandler im Konstruktor:

```
        public App()
        {
            this.Startup += this.Application_Startup;
            this.Exit += this.Application_Exit;
            this.UnhandledException += this.Application_UnhandledException;
            InitializeComponent();
        }
```

Hier können Sie auf den Start der Silverlight-Anwendung reagieren, standardmäßig wird hier die anzuzeigende Seite festgelegt:

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            this.RootVisual = new MainPage();
        }
```

Auch auf das Ende Ihrer Anwendung können Sie reagieren:

```
        private void Application_Exit(object sender, EventArgs e)
        {       }
```

Eine erste Fehlerbehandlung ist bereits "ab Werk" mit dabei:

```
        private void Application_UnhandledException(object sender,
                                    ApplicationUnhandledExceptionEventArgs e)
        {
...
```

BEISPIEL 43.2: Beispiel für App.xaml & App.xaml.cs

```
            if (!System.Diagnostics.Debugger.IsAttached)
            {
...
                e.Handled = true;
                Deployment.Current.Dispatcher.BeginInvoke(delegate { ReportErrorToDOM(e); });
            }
        }

        private void ReportErrorToDOM(ApplicationUnhandledExceptionEventArgs e)
        {
            try
            {
                string errorMsg = e.ExceptionObject.Message + e.ExceptionObject.StackTrace;
                errorMsg = errorMsg.Replace('"', '\'').Replace("\r\n", @"\n");
                System.Windows.Browser.HtmlPage.Window.Eval
                      ("throw new Error(\"Unhandled Error in Silverlight Application " +
                        errorMsg + "\");");
            }
            catch (Exception)
            { }
        }
    }
}
```

Startup

Standardmäßig wird hier die Startseite zugewiesen:

HINWEIS: Haben Sie in obigem *Application_Startup* eine Startseite per *RootVisual* einmal zugewiesen, können Sie dies nicht mehr ändern. Eine erneute Zuweisung ist nicht zulässig.

Nutzen Sie das *Startup*-Ereignis weiterhin, um Initialisierungen zu realisieren oder die Übergabeparameter auszuwerten.

BEISPIEL 43.3: Auswerten optionaler Übergabeparameter an das Plug-in

C#

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            this.RootVisual = new MainPage();
            foreach (var param in e.InitParams)
              MessageBox.Show(param.Key + " = " + param.Value);
        }
```

Nach Ausführung des *Startup*-Ereignisses wird die Silverlight-Anwendung sichtbar.

Exit

Das *Exit*-Ereignis feuert beim Beenden der Anwendung. Folgende Auslöser sind möglich:

- Schließen des Browsers
- Schließen des Browser-Tabs, in dem die Silverlight-Anwendung angezeigt wird
- Navigation auf eine andere Webseite
- Entfernen des HTML-Elements mit dem Silverlight Plug-in aus der Webseite

UnhandledException

Hier können/sollten Sie Fehler behandeln, die innerhalb Ihrer Anwendung auftreten, aber nicht behandelt wurden. Wird der Fehler behandelt, so signalisieren Sie dies mit der *Handled*-Eigenschaft.

HINWEIS: Behandeln Sie den Fehler nicht oder belassen Sie *Handled* auf *false*, wird die Anwendung beendet und das *onError*-Ereignis des Plug-in wird ausgelöst (JavaScript). In diesem Fall wird kein *Exit*-Ereignis ausgelöst.

Weitere Eigenschaften und Methoden der Application-Klasse

Sicher nicht ganz zufällig wird im Rahmen der Silverlight-Anwendung auch ein Interesse an den Einstellungen/Parametern des umgebenden Silverlight-Plug-in bestehen. An dieser Stelle hilft Ihnen die *Application*-Klasse weiter. Über das *Current.Host*-Objekt rufen Sie die gewünschten Einstellungen (*Background, Source, Settings, Content ...*) ab.

HINWEIS: Weitere Methoden stehen im Zusammenhang mit dem Erstellen und Warten von Out-Of-Browser-Anwendungen zur Verfügung (siehe dazu 43.8.1).

43.3.2 MainPage.xaml & MainPage.xaml.cs

Hier haben Sie es mit der standardmäßig angezeigten Seite Ihrer Silverlight-Anwendung zu tun:

- Ihre Anwendung kann mehrere Seiten enthalten, ein direkter Wechsel zwischen diesen ist jedoch nicht möglich.
- *UserControls*, d h. auch diese Seite, können wiederum andere *UserControls* enthalten. Sie können die Hauptseite also als Container für weitere *UserControls* verwenden.

Konzeptionell können Sie diese Dateien mit den *Window*-Dateien von WPF vergleichen, auch wenn es sich in diesem Fall um eine Instanz der *UserControl*-Klasse handelt.

BEISPIEL 43.4: Eine leere Seite

XAML
```
<UserControl x:Class="SilverlightApplication4.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

BEISPIEL 43.4: Eine leere Seite

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

C#

```
...
namespace Beispiel
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

Mehr zu diesen Dateien finden Sie im Abschnitt 43.4 ab Seite 2127.

43.3.3 AssemblyInfo.cs

Wie auch bei den Windows Forms- oder WPF-Anwendungen werden hier die Assembly-Informationen (*Title, Description, Company, Version* etc.) gespeichert.

```
...
[assembly: AssemblyTitle("Beispiel")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
...
```

Die Einträge können Sie auch über den Menüpunkt *Projekt|Eigenschaften|Silverlight|Assemblyinformationen* komfortabel bearbeiten.

43.4 Fenster und Seiten in Silverlight

Im Gegensatz zu einer optisch ähnlichen WPF-Anwendung kann Silverlight eigentlich immer nur ein "Fenster" zur gleichen Zeit anzeigen. Prinzipiell ist es zunächst auch nicht möglich, dieses "Fenster" durch ein anderes zu ersetzen. Wie es dennoch geht, wie Sie Popup-Fenster einblenden und wie Sie zwischen Seiten navigieren können, wollen wir im Folgenden klären.

43.4.1 Das Standardfenster

Eigentlich handelt es sich nicht um ein Fenster, sondern um eine Instanz der Klasse *UserControl*. Da das weitere Verhalten aber einem Fenster entspricht, wollen wir auch bei dieser Bezeichnung bleiben.

Wie schon erwähnt, kann eine Silverlight-Anwendung zwar über mehrere Fenster verfügen, ein nachträgliches (nach der erstmaligen Zuweisung von *RootVisual* in *App.xaml.cs*) Austauschen ist jedoch nicht möglich.

Drei Lösungsmöglichkeiten für dieses Problem bieten sich an:

- eine etwas unkonventionelle Lösung durch Zuweisen eines neuen *Content*
- die Verwendung von Popup-Fenstern der Klasse *ChildWindow*
- der grundsätzliche Entwurf als Navigationsanwendung

Ein kleines Beispiel soll die erste Lösung demonstrieren. Diese ist jedoch konzeptionell auf wenige Seiten beschränkt, andernfalls verlieren Sie recht schnell die Übersicht in Ihrem Projekt.

BEISPIEL 43.5: Neue Seite per Content zuweisen

C#

Erstellen Sie eine Hauptseite *MainPage.xaml* und zusätzlich eine weitere Seite (*Silverlight-Control1.xaml*). Fügen Sie in beide Seiten jeweils einen Button ein und weisen Sie die folgenden Ereignisprozeduren zu:

In *MainPage.xaml.cs*:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo Silverlight-Programmierer! Es folgt Seite 2");
    this.Content = new SilverlightControl1();
}
```

In *SilverlightControl1.xaml.cs*:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    this.Content = new MainPage();
}
```

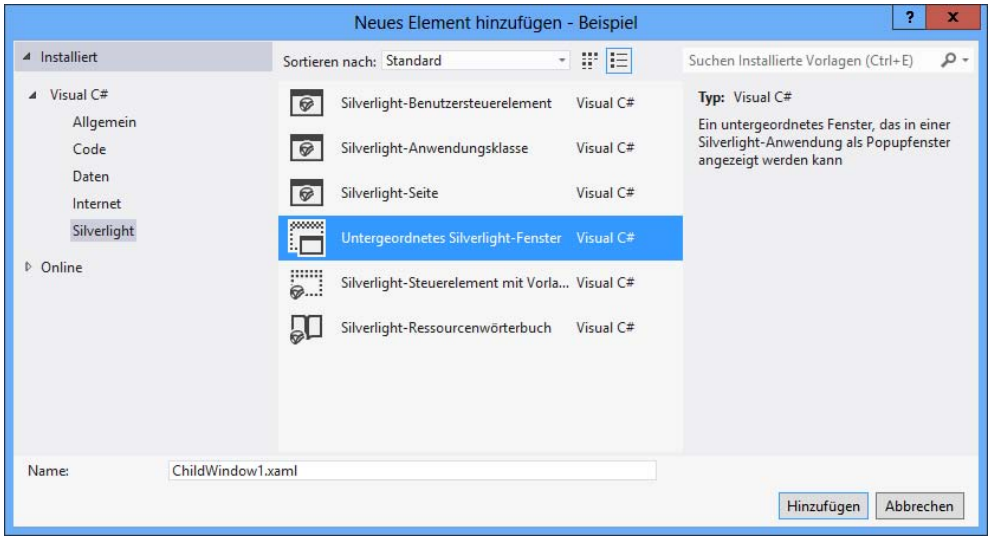
So einfach geht das! Über den Garbage-Collector werden die überflüssigen Instanzen auch wieder entsorgt.

Wie Sie in einem Fenster mittels Layout-Controls weitere Steuerelemente anordnen brauchen wir an dieser Stelle sicher nicht noch einmal zu wiederholen, da können wir Sie problemlos auf Abschnitt Fehler: Referenz nicht gefunden, Seite Fehler: Referenz nicht gefunden, verweisen.

43.4.2 Untergeordnete Silverlight-Fenster

Möchten Sie Statusmeldungen anzeigen oder Eingaben separat vornehmen ist es sinnvoll, mit einem extra Fenster zu arbeiten, das aus dem standardmäßig angezeigtem Fenster heraus eingeblendet wird. Dazu steht Ihnen in Silverlight die Klasse *ChildWindow* zur Verfügung.

Sie erzeugen ein derartiges Fenster, indem Sie dem Projekt ein neues Element "Untergeordnetes Silverlight-Fenster" hinzufügen:



Das Fenster verfügt bereits über zwei Schaltflächen (OK und Abbruch) mit den entsprechenden Ereignishandlern. Weitere Inhalte können Sie nach Belieben einfügen.

BEISPIEL 43.6: Ein untergeordnetes Fenster einblenden

C#

Mit dem Klick auf eine Schaltfläche:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
```

Definieren:

```
    ChildWindow1 cwin = new ChildWindow1();
    cwin.Closed += new EventHandler(cwin_Closed);
```

Einblenden:

```
    cwin.Show();
```

BEISPIEL 43.6: Ein untergeordnetes Fenster einblenden

Die Auswertung kann nur per Ereignishandler erfolgen, da das Fenster zwar optisch als modal fungiert, die *Show*-Methode aber nicht blockiert und die Ausführung sofort wieder fortsetzt:

```
void cwin_Closed(object sender, EventArgs e)
{
```

Hier müssen wir zunächst die Instanz abrufen:

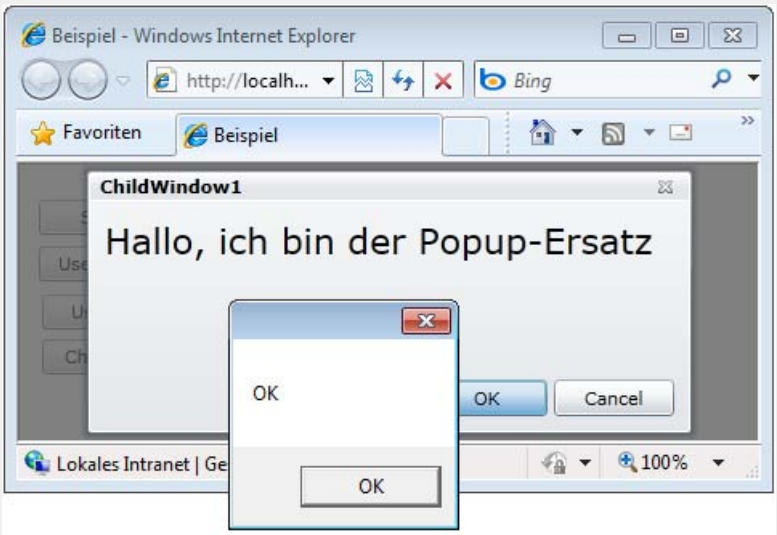
```
    ChildWindow1 cwin = (ChildWindow1)sender;
```

Auswertung:

```
    if (cwin.DialogResult == true)
        MessageBox.Show("OK");
    else
        MessageBox.Show("Abbruch");
}
```

Ergebnis

Das folgende Fenster wird mit einer Animation eingeblendet:



Wie Sie sehen, wird zwar das *Closed*-Ereignis korrekt ausgewertet, leider ist aber zu diesem Zeitpunkt noch das *ChildWindow* sichtbar.

HINWEIS: Während das *ChildWindow* geöffnet ist kann kein Zugriff auf das darunterliegende Silverlight-Control erfolgen. Der Hintergrund wird grau ausgeblendet[1], um das *ChildWindow* hervorzuheben.

[1] Fensterhintergrund, Rahmen, Button etc. sind alles Templates, die zum *ChildWindow* gehören. Bei Bedarf können Sie diese natürlich auch anpassen.

43.4.3 UserControls für die Anzeige von Detaildaten

Nicht immer können Sie in einem Fenster alle gewünschten Inhalte darstellen. Auch die Verwendung des *TabControls* führt bei einer großen Anzahl von Tabs recht schnell zu unübersichtlichen Programmen. Besser Sie verwenden in diesem Fall ein *UserControl*, das mit einer geeigneten Schnittstelle versehen seine Funktionalität weitgehend kapseln kann.

Der Entwurf erfolgt in einem eigenen Fenster, so leidet auch die Übersicht nicht darunter. Verfügt Ihre Anwendung über sehr viele UserControls, können Sie diese optional auch in eine externe Assembly auslagern und später dynamisch nachladen. So wird die Größe des Anwendungspackage zunächst verringert.

Ein-/Ausblenden von UserControls

Geht es lediglich darum, Inhalte eines UserControls ein- bzw. auszublenden, können Sie das gewünschte Control bereits zur Entwurfszeit per Drag & Drop aus dem Werkzeugkasten in die gewünschte Seite einfügen[1]. Soll es bei Start der Anwendung nicht sichtbar sein, setzen Sie per XAML die Eigenschaft *Visibility* einfach auf *Collapsed.*

BEISPIEL 43.7: Ausblenden eines UserControls

XAML
```
<my:SilverlightControl2 HorizontalAlignment="Left" Margin="204,127,0,0"
        x:Name="silverlightControl21" VerticalAlignment="Top" Height="152" Width="234"
        Visibility="Collapsed" />
```

Zu Laufzeit können Sie das Control dann über diese Eigenschaft ein- und wieder ausblenden.

BEISPIEL 43.8: *UserControl* zur Laufzeit einblenden

C#
```
private void button2_Click(object sender, RoutedEventArgs e)
 {
     silverlightControl21.Visibility = System.Windows.Visibility.Visible;
 }
```

Hinzufügen/Entfernen von UserControls

Eine weitere Verwendungsmöglichkeit der UserControls bietet sich mit dem dynamischen Hinzufügen zur Laufzeit. Beispielsweise können Sie ein *StackPanel* in Ihre Hauptseite als Container einfügen. Zur Laufzeit nutzen Sie die *Children.Add*-Methode, um UserControls im *StackPanel* anzuzeigen.

BEISPIEL 43.9: *UserControl* dynamisch zur Laufzeit erzeugen und einblenden

C#
```
private void button3_Click(object sender, RoutedEventArgs e)
{
    stackPanel1.Children.Add(new SilverlightControl2());
}
```

[1] Falls das Control nicht sichtbar ist, das Projekt bitte kompilieren!

HINWEIS: Entfernen können Sie das *UserControl* durch Aufruf der *Children.Clear*-Methode.

43.4.4 Echte Windows

Mit der Silverlight-Version 5 hatten die Entwickler endlich ein Einsehen – statt der bisherigen Notbehelfe *PopUp* und *ChildWindow* gibt es jetzt endlich auch ein echtes *Window*.

HINWEIS: Die Fenster stehen nur Out-of-Browser-Anwendungen zur Verfügung, die im Full-Trust Modus laufen.

Grundsätzlich können Sie mit der Klasse *Window* ein/mehrere neue, frei positionierbare Fenster erzeugen, die weitgehend unabhängig vom Hauptfenster agieren. Wie bei Windows Forms- oder WPF-Anwendungen üblich, werden diese Fenster mit dem Schließen des Hauptfensters ebenfalls geschlossen, um die Anwendung zu beenden.

HINWEIS: Leider müssen Sie derartige Fenster zunächst mit C#-Code erzeugen, es steht kein XAML-Editor zur Verfügung.

Diese Einschränkung lässt sich jedoch leicht durch Verwendung eines UserControls für den Content des *Window* umgehen.

BEISPIEL 43.10: Erzeugen eines *Window*

C#

```
...
```

Instanz erzeugen:

```
Window win = new Window();
```

Skalieren:

```
win.Width = 450;
win.Height = 320;
```

Beschriften:

```
win.Title = "Ein frei schwebendes Fenster";
win.WindowState = WindowState.Normal;
```

Sichtbarkeit festlegen:

```
win.TopMost = true;
```

Anzeige:

```
win.Show();
```

```
...
```

Das alles sieht schon viel freundlicher aus, als die Quälereien mit *ChildWindow*.

Ein- und Ausblenden können Sie derartiger Fenster mit *Show* und *Hide*, mit *Activate* holen Sie das Fenster im Zweifelsfall wieder in den Vordergrund. Ist die Eigenschaft *WindowState* auf *Normal* festgelegt, können Sie das *Window* mit den Eigenschaften *Left* und *Top* frei positionieren, einen sinnvollen Überblick über die verfügbaren Bildschirme haben Sie jedoch nach wie vor nicht.

Wem die Optik des Fensters nicht gefällt (Fensterschaltflächen, Rahmenbreite), der hat die Möglichkeit, den Fensterrahmen über die Eigenschaft *WindowStyle* auszublenden (*WindowStyle.None*). In diesem Fall müssen Sie sich jedoch um eigene Rahmen, Fensterschaltflächen und Bewegungsmöglichkeiten per Drag & Drop selbst kümmern.

Damit Sie bei diesem Überangebot an Fenstern nicht den Überblick verlieren, stellt die Collection *Application.Current.Windows* eine Liste aller geöffneten Fenster zur Verfügung.

Mehr zu *Window*-Klasse finden Sie in unserem Praxisbeispiel

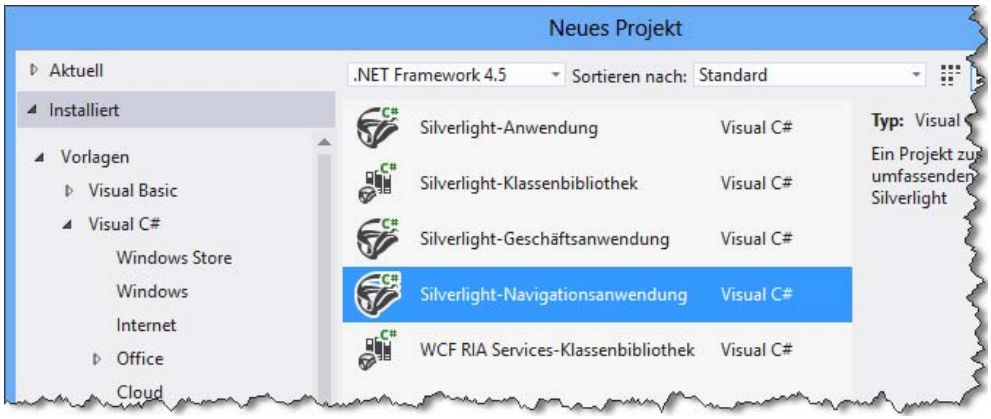
► 43.8.12 Weitere Fenster öffnen

43.4.5 Navigieren in Silverlight-Anwendungen

Wenn Ihnen die im Abschnitt 43.4.1 gezeigte Lösung nicht genügt bzw. zu unübersichtlich ist, kommen Sie früher oder später auch mit dem Navigations-Framework in Kontakt. Dieses bietet Ihnen eine einfache und intuitive Möglichkeit, ausgehend von einer Startseite (diese stellt auch das Master-Layout) einzelne Pages einzublenden bzw. diese gegen andere Pages auszutauschen.

Projekt erzeugen

Der einfachste Weg zu einer derartigen Anwendung führt über das Template "Silverlight-Navigationsanwendung":



HINWEIS: In diesem Fall brauchen Sie sich auch nicht um die zusätzlichen Verweise auf die Assemblies *System.Windows.Controls* und *System.Windows.Controls.Navigation* zu kümmern.

Ganz nebenbei wird bereits eine Hauptseite (in diese werden die anderen Seiten eingeblendet) und zwei Detailseiten (*Home*, *About*) sowie eine Fehlerseite (als *ChildWindow*) erzeugt.

Bereits diese kleine Anwendung zeigt, dass Instanzen von *Page* (die Detailseiten) als Ziele der Navigation eines *Frame*-Controls genutzt werden. Zusätzlich bietet sich die Möglichkeit, per Uri-Mapping besser lesbare Uris zu verwenden. Doch dazu später mehr.

Die Page-Klasse

Zunächst wollen wir uns kurz die jeweiligen Detailseiten ansehen, die als Instanzen der dafür zuständigen *Page*-Klasse angelegt werden. Die Vorgehensweise beim Zusammenstellen der Seite entspricht der bei den bereits vorgestellten UserControl-Seiten, auch hier können Sie, ausgehend von einem Layout-Control, die Inhalte Ihrer Seite zusammenstellen.

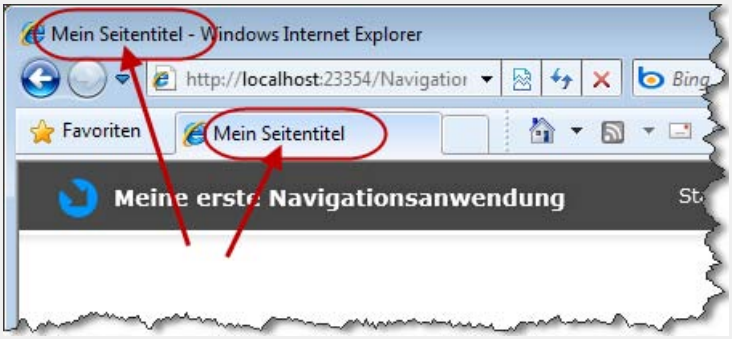
Ein Blick auf die Eigenschaften von *Page* zeigt, dass auch eine *Title*-Eigenschaft vorhanden ist, die auch sinnvoll genutzt werden kann:

BEISPIEL 43.11: Verwendung von *Title*

XAML

```
<navigation:Page x:Class="Navigation.Page1"
...
           d:DesignWidth="640" d:DesignHeight="480"
           Title="Mein Seitentitel">
```

Ergebnis



Drei Methoden sollten Sie sich ebenfalls merken, um diese gegebenenfalls zu überschreiben und deren Funktionalität zu nutzen:

- *OnNavigatedFrom*
  (wird aufgerufen, wenn diese Page **nicht** mehr die aktive Seite ist)
- *OnNavigatedTo*
  (wird aufgerufen, wenn diese Page die aktive Seite wird)
- *OnNavigatingFrom*
  (wird aufgerufen, bevor diese Page nicht mehr die aktive Seite ist, auch ein Abbruch der Navigation über *e.Cancel* ist möglich)

An dieser Stelle kommt sicher auch die Frage auf, wie aus der Seite heraus navigiert werden kann. Dazu steht Ihnen über die *NavigationService*-Eigenschaft der *Page* das Navigations-Framework zur Verfügung.

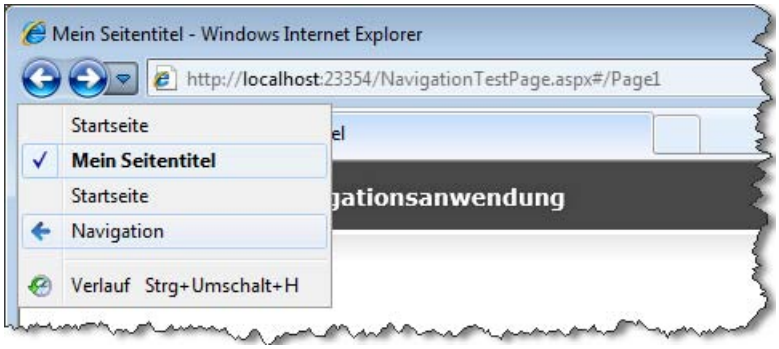
BEISPIEL 43.12: Auf die vorhergehende Seite navigieren

C#
```
private void button1_Click(object sender, RoutedEventArgs e)
{  this.NavigationService.GoBack(); }
```

Zur Verfügung stehen Ihnen die Eigenschaften *CanGoBack* und *CanGoForward* sowie *CurrentSource*, mit denen Sie abfragen können, ob eine Navigation möglich ist bzw. um welche Seite es sich aktuell handelt.

Die eigentliche Navigation lösen Sie mit *GoBack*, *GoForward* und *Navigate* aus. Zusätzlich stehen in diesem Zusammenhang auch noch diverse Ereignisse zur Auswertung der Navigation zur Verfügung.

Ganz nebenbei funktionieren in diesem Zusammenhang auch die Navigationstasten Ihres Web-Browsers, wie die folgende Abbildung der Browser-Historie zeigt:



HINWEIS: Möchten Sie unerwünschte Navigationen (z.B. Zurück) über die Browser-Tasten ausschließen, müssen Sie die *OnNavigatingFrom*-Methode überschreiben und auswerten, welche Seite angesteuert wird. Ist dies nicht zulässig, brechen Sie hier einfach die Navigation ab.

Der Navigations-Frame

Damit kommen wir noch einmal zum Haupt-Akteur im Navigations-Framework zurück, d h. den *Frame* in der Hauptseite Ihrer Anwendung, der die einzelnen Pages aufnehmen soll.

HINWEIS: Alles was sich auf der Hauptseite Ihrer Anwendung befindet (Navigationsschaltflächen, Titelbänder etc.) bleibt bei einem Seitenwechsel erhalten, es wechselt nur der Inhalt der Frame-Elements!

Der *Frame* selbst verfügt über einige interessante Eigenschaften und Methoden, die im Zusammenhang mit der Navigation von Bedeutung sind:

- Die Methoden *GoBack*, *GoForward* und *Navigate* dürften Ihnen bereits aus dem vorhergehenden Abschnitt bekannte sein,
- was auch auf die Eigenschaften *CanGoBack* und *CanGoForward* zutrifft.
- Die Auswertung können Sie mit den Ereignissen *Navigated*, *Navigating NavigationFailed* und *NavigationStopped* vornehmen.

HINWEIS: Die Ereignisse *NavigationFailed* und *Navigated* sind bereits implementiert, um die notwendigsten Funktionalitäten bereitzustellen.

BEISPIEL 43.13: Die bereits implementierten Ereignisse zum Navigieren

C#

Nach dem Einblenden einer neuen Seite wird für die Hyperlink-Schaltflächen sichergestellt, dass auch der richtige Button markiert ist:

```
private void ContentFrame_Navigated(object sender, NavigationEventArgs e)
{
    foreach (UIElement child in LinksStackPanel.Children)
    {
        HyperlinkButton hb = child as HyperlinkButton;
        if (hb != null && hb.NavigateUri != null)
        {
            if (hb.NavigateUri.ToString().Equals(e.Uri.ToString()))
                VisualStateManager.GoToState(hb, "ActiveLink", true);
            else
                VisualStateManager.GoToState(hb, "InactiveLink", true);
        }
    }
}
```

Kleinere Fehler werden durch die Anzeige eines *ChildWindow* gemeldet:

```
private void ContentFrame_NavigationFailed(object sender,NavigationFailedEventArgs e)
{
    e.Handled = true;
    ChildWindow errorWin = new ErrorWindow(e.Uri);
```

BEISPIEL 43.13: Die bereits implementierten Ereignisse zum Navigieren

```
            errorWin.Show();
        }
```

Eine direkte Navigation per Methode könnte zum Beispiel wie folgt aussehen:

BEISPIEL 43.14: Navigieren per *Frame*

C#

Haben Sie eine Schaltfläche in *MainPage.xaml* eingefügt, kann der folgende Ereignishandler verwendet werden:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    this.ContentFrame.Navigate(new Uri("/Page1", UriKind.Relative));
}
```

Ein Blick in die Frame-Definition (*MainPage.xaml*) dürfte ebenfalls recht aufschlussreich sein, werden doch hier die Zusammenhänge zwischen angeforderter und angesteuerter Uri mittels Uri-Mapping hergestellt:

BEISPIEL 43.15: Uri-Mapping im Frame

XAML

```
<Border x:Name="ContentBorder" Style="{StaticResource ContentBorderStyle}" >
    <navigation:Frame x:Name="ContentFrame"
     Style="{StaticResource ContentFrameStyle}"  Source="/Home"
       Navigated="ContentFrame_Navigated"
       NavigationFailed="ContentFrame_NavigationFailed">
        <navigation:Frame.UriMapper>
          <uriMapper:UriMapper>
            <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml"/>
            <uriMapper:UriMapping Uri="/{pageName}" MappedUri="/Views/
                                        {pageName}.xaml"/>
          </uriMapper:UriMapper>
        </navigation:Frame.UriMapper>
    </navigation:Frame>
</Border>
```

Sicher haben Sie auch schon bemerkt, dass sich die Detailseiten beim erzeugten Projekt im Unterverzeichnis *Views* befinden. Zum einen ist es recht aufwändig, immer den vollen Uri für dieses Verzeichnis zu verwenden, zum anderen führt dies zu einer recht unübersichtlichen Angabe in der Adresszeile des Browsers. Aus diesen Gründen wird mittels Uri-Mapping eine übergebene Adresse (*Uri*) so "gemappt", dass zur gewünschten Adresse (*MappedUri*) gesprungen wird.

Im obigen Beispiel wird bei fehlender Frame-Adresse (*Uri* = *""*)

```
http://localhost:23354/NavigationTestPage.aspx
```

die Seite */Views/Home.xaml* angezeigt:

```
http://localhost:23354/NavigationTestPage.aspx#/Home
```

Im Zusammenhang mit der Detaildatenanzeige bei Datenbankanwendungen bietet sich hier auch die Möglichkeit, den Primärkey für die Navigation zwischen den Seiten zu nutzen. Dazu wird die Adresse entsprechend umgeformt, sodass ein auslesbarer QueryString entsteht:

BEISPIEL 43.16: QueryString per Mapping erzeugen

XAML
```
...
   <uriMapper:UriMapping Uri="Artikel/{id}" MappedUri="/Views/ArtikelDetails.xaml?id={id}"/>
...
```

Aus der eingegebenen Adresse

```
http://testweb.de/Warenkorb.asp#/Artikel/27
```

wird somit

```
http://testweb.de/Warenkorb.asp#/Views/ArtikelDetails.xaml?id=27
```

Die Id können Sie jetzt per QueryString in *ArtikelDetails.xaml* mit der folgenden Anweisung abrufen:

```
this.NavigationContext.QueryString["id"]
```

43.5 Datenbanken/Datenbindung

Mit dem Ende des letzten Abschnitts sind wir bereits mitten in der Thematik "Datenbanken/Datenbindung" angekommen. Allerdings werden wir uns hier nicht großartig wiederholen (siehe dazu Kapitel 15, sondern uns auf einige wesentliche Besonderheiten im Zusammenhang mit Silverlight-Anwendungen beschränken.

Zunächst ein wichtiger Hinweis vorweg:

HINWEIS: Grundsätzlich sollten Sie immer das Konzept von Silverlight im Hinterkopf behalten: Der Code wird immer auf dem Client ausgeführt. Dies führt auch dazu, dass Sie nicht auf vermeintlich lokale Datenbanken (wie bei ASP.NET-Anwendungen) zugreifen können. Diese befinden sich aus Sicht der Anwendung immer auf einem externen System, auf das Sie nur über das Web zugreifen können.

Eine der wenigen Ausnahmen von obiger Aussage ist die Verwendung des isolierten Speichers für eine lokale Datenbank auf dem Client, wie zum Beispiel im Projekt "Silverlight Database", das Sie unter der folgenden Adresse herunterladen können:

LINK: http://silverdb.codeplex.com/

Für alle anderen Fälle gilt: die Datenbank liegt auf dem Server und ist damit aus Sicht des Clients "weit weg". Womit auch schon klar wird, dass wir auf Datenbanken immer nur mit zusätzlichen Services (Proxys) zugreifen können. Derzeit stehen Ihnen folgende sinnvolle Varianten zur Verfügung:

- die "guten alten" ASP.NET Webdienste,
- WCF Dienste (auch Duplex)
- und WCF Data Services (die ehemaligen ADO.NET Data Services).

Zusätzlich können Sie natürlich auch andere Datenquellen "anzapfen", wie zum Beispiel mit der *HttpWebRequest*-Klasse auf RSS-Feeds oder per *Socket*-Klasse auf die TCP-Sockets zugreifen, aber das soll im Rahmen dieses Buchs kein Thema sein.

Im Weiteren wollen wir uns auf die Anbindung eines Webdienstes und den Zugriff auf die WCF Data Services beschränken und die Vorgehensweise mit zwei umfangreicheren Beispielen demonstrieren.

HINWEIS: Da die Vorgehensweise bei den beiden o.g. Dienstformen weitgehend übereinstimmt, werden wir diese auch gemeinsam vorstellen.

HINWEIS: Auf ausführliche Darstellungen zu den Themen "WCF-Dienste" und "WCF-Data Services" müssen wir aus Platzgründen an dieser Stelle verzichten, siehe dazu unser Buch [Datenbankprogrammierung mit Visual C# 2012].

43.5.1 ASP.NET-Webdienste/WCF-Dienste

Ausgangspunkt unserer Betrachtungen ist ein bereits erstelltes Silverlight-Projekt mit dazugehörigem ASP.NET-Anwendungsprojekt.

Folgende Schritte werden wir abarbeiten:

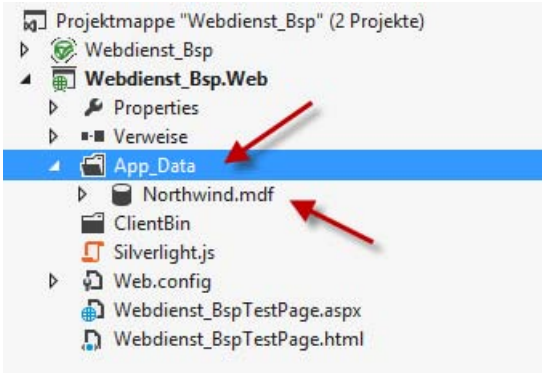
- Einfügen einer SQL Server-Datenbank
- Erstellen eines EDM-Datenmodells
- Erstellen des Webdienstes/WCF Dienstes
- Einbinden der Dienste in das Silverlight-Projekt
- Konfigurieren der Silverlight-Oberfläche und Abrufen der Daten

Und damit auf in die Praxis ...

Einfügen der SQL Server-Datenbank

Erster Schritt ist natürlich das Erstellen einer Datenbasis. Auf die Schritte zum Erstellen einer neuen Datenbank verzichten wir, stattdessen nutzen wir die Beispieldatenbank *Northwind.mdf*, die Sie auch in den Buch-Beispieldaten finden.

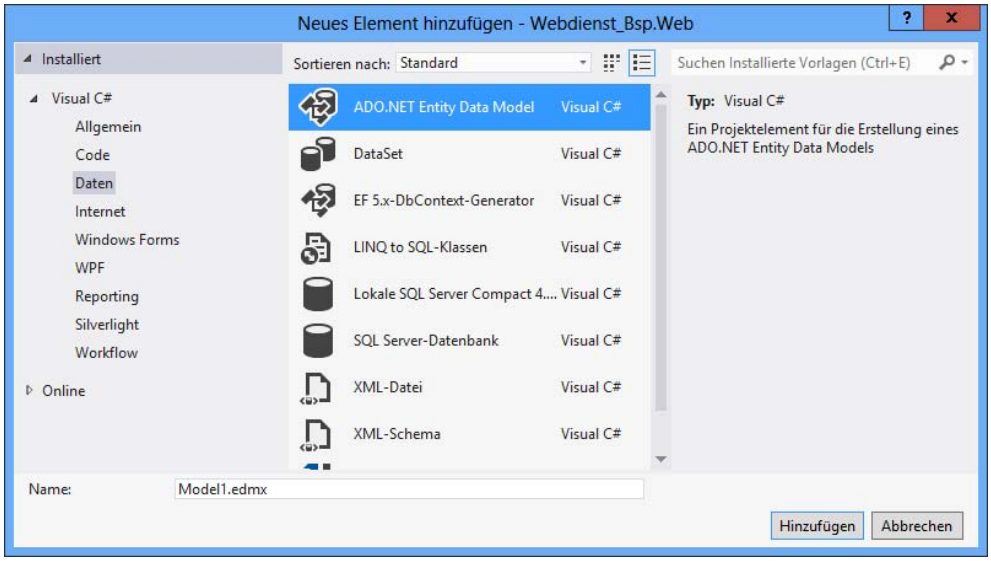
Erstellen Sie zunächst einen neuen *App_Data*-Ordner im ASP.NET-Web-Projekt (nicht im Silverlight-Projekt), indem Sie im Projektmappen Explorer mittels Kontextmenü den Eintrag *Hinzufügen|Hinzufügen ASP.NET Ordner|App_Data* aufrufen. In den neu erstellten Ordner ziehen Sie einfach die Datei *Northwind.mdf* per Drag & Drop hinein:



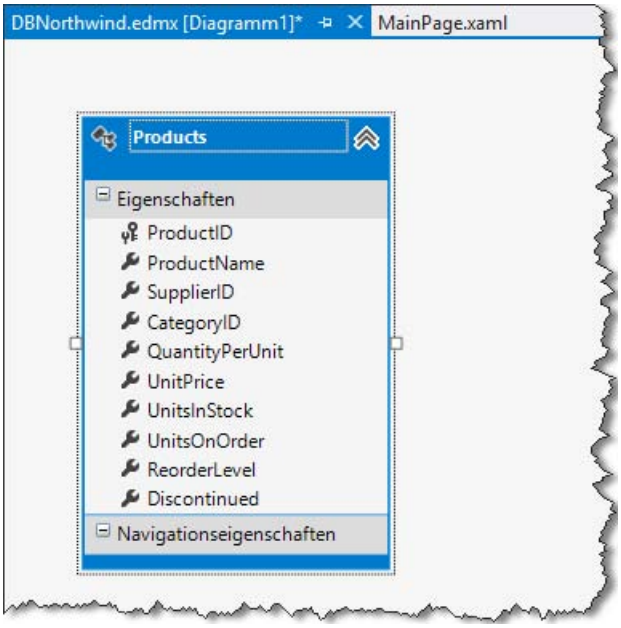
Damit ist der "Datenbank-Import" abgeschlossen und wir können uns um das Datenmodell kümmern.

Erstellen des EDM-Datenmodells

Wie auch im folgenden Abschnitt wollen wir mit einem Entity-Datenmodell arbeiten. Wählen Sie den Menüpunkt *Hinzufügen|Neues Element hinzufügen* aus dem Kontextmenü des Projektmappen Explorers (für das ASP.NET-Projekt). Im folgenden Fenster (siehe nächste Abbildung) markieren Sie die Rubrik "Daten" und können jetzt die Vorlage "ADO.NET Entity Data Model" auswählen. Vergeben Sie den Namen *dbNorthwind.edmx*.



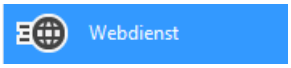
Wählen Sie im folgenden Assistenten als Datenquelle unsere *Northwind*-Datenbank (Option *Aus Datenbank generieren*) aus und fügen Sie die Tabelle *Products* in das Datenmodell ein:



Visual Studio hat jetzt im Hintergrund entsprechende Mapperklassen erstellt, mit denen Sie auf die SQL Server Datenbank *Northwind* und die Tabelle *Products* zugreifen können.

Erstellen des ASP.NET Webdienstes

Zunächst wollen wir uns auf die schon lange eingeführten Webdienste beschränken, diese sind weit verbreitet und verzichten auf unnötigen Konfigurationsaufwand. Fügen Sie dem ASP.NET-Testprojekt einen Webdienst hinzu, indem Sie ebenfalls über das Kontextmenü die Liste der Templates öffnen. In der Rubrik "Internet" finden Sie die altbekannten Webdienste:



Belassen Sie es einfach bei der vorgeschlagenen Namensgebung und öffnen Sie nachfolgend die neu erstellte Datei *Webservice1.asmx.cs*. Fügen Sie die im Folgenden fett hervorgehobene Methode ein:

```
...
using System.Web.Services;

namespace Daten1.Web
{
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    public class WebService1 : System.Web.Services.WebService
    {
```

```
...
```

Eine Webmethode erstellen, die eine *List* von *Products*-Objekten zurückgibt:

```
        [WebMethod]
        public List<Products> GetAllProducts()
        {
```

Verbindung zur Datenbank aufnehmen:

```
            DBNorthwindDataContext db = new DBNorthwindDataContext();
```

Daten abrufen und zurückgeben:

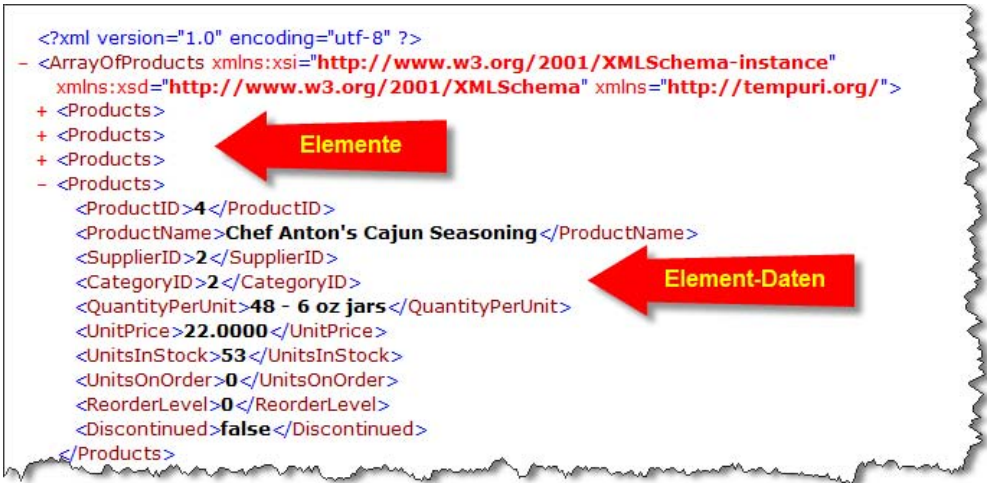
```
            var prod = db.Products;
            return prod.ToList();
        }
    }
}
```

Damit ist unser Webdienst bereits einsatzbereit. Sie können das Projekt testweise starten, indem Sie in der Adresszeile statt des Namens der ASPX-Datei unseren *Webservice1.asmx angeben.*

Nachfolgend erscheint die Dienstbeschreibung, unser Webservice enthält neben der neuen Methode *GetAllProducts* auch noch das standardmäßige *HelloWorld*-Beispiel:

Ein Klick auf *GetAllProducts* öffnet die genaue Methodenbeschreibung mit Anforderungsbeispielen. Über die eingeblendete Schaltfläche "Aufrufen" können Sie einen ersten Testlauf starten (siehe folgende Abbildung).

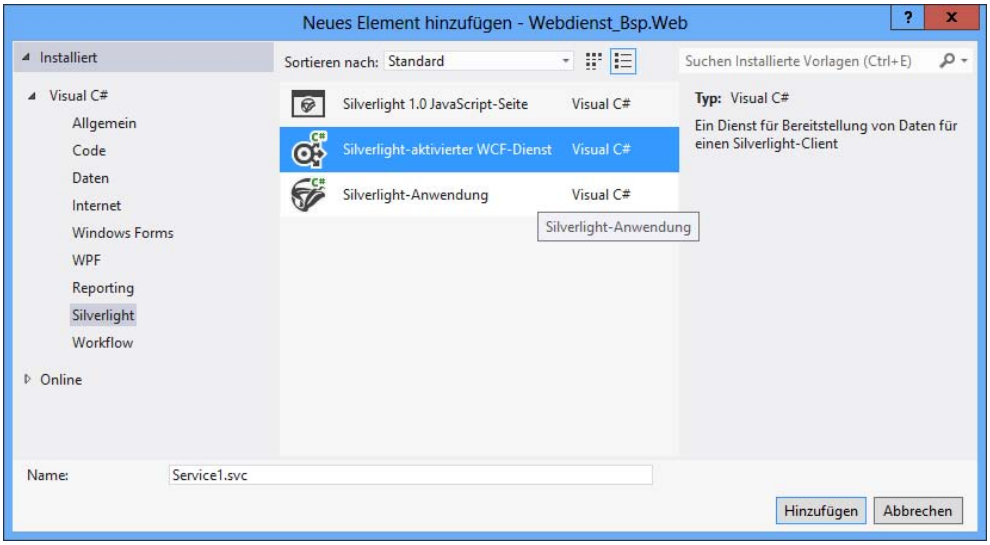
Sollten Sie hier keine Daten angezeigt bekommen, ist vermutlich bei der Konfiguration Ihrer Serververbindung etwas faul, lösen Sie zunächst dieses Problem, bevor Sie sich der Einbindung in das Silverlight-Projekt widmen.



Doch bevor es so weit ist, wollen wir noch kurz die Alternative zum Webdienst, d h. dessen jüngeren Bruder betrachten:

Alternative: Erstellen eines WCF Dienstes

Auch hier rufen Sie zunächst über den Projektmappen-Explorer die Liste der verfügbaren Templates ab. Wichtig ist hier, dass Sie nicht einfach einen WCF Dienst auswählen, sondern Sie müssen den Spezialtyp "Silverlight-aktivierter WCF-Dienst" aus der Rubrik "Silverlight" auswählen:



HINWEIS: Sollten Sie nur einen "gewöhnlichen" WCF-Dienst erzeugen, werden spätere Verbindungsversuche fehlschlagen, Silverlight kann sich nur mit SOAP 1.1 Diensten verbinden.

Öffnen Sie die erstellte Datei *Service1.svc.cs* und fügen Sie die fett hervorgehobenen Zeilen hinzu:

```
...
using System.Collections.Generic;
namespace Daten1.Web
{
    [ServiceContract(Namespace = "")]
    [AspNetCompatibilityRequirements(RequirementsMode =
             AspNetCompatibilityRequirementsMode.Allowed)]
    public class Service1
    {
...
```

In WCF wird mit Contracts gearbeitet:

```
        [OperationContract]
        public List<Products> GetAllProducts()
```

Auch hier geben wir eine Liste von *Products*-Objekten zurück, der Datenabruf per EDM-Klassen entspricht der Webdienst-Version:

```
        {
            DBNorthwindDataContext db = new DBNorthwindDataContext();
            var prod = db.Products;
            return prod.ToList();
        }
    }
}
```

Auch hier können Sie einen Test starten, allerdings werden Ihnen in diesem Fall keine Daten angezeigt, eine Prüfung der Funktionsfähigkeit ist also nicht möglich.

Einbinden der Dienste in das Silverlight-Projekt

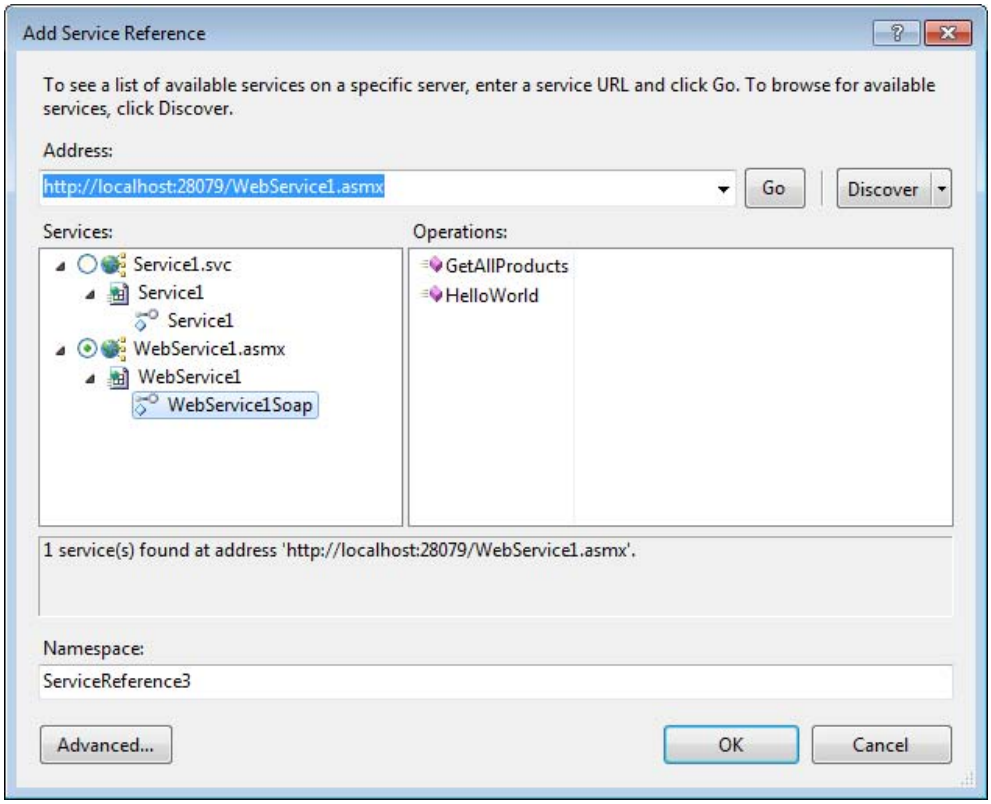
Damit sind wir auch schon bei unserem Endverbraucher, dem Silverlight-Projekt angelangt. Erster Schritt ist in jedem Fall das Einbinden der Dienstverweise (*Projekt|Dienstverweis hinzufügen*).

HINWEIS: Sie können an dieser Stelle natürlich auch beide Dienste einbinden.

Da sich beide Dienste in der gleichen Projektmappe befinden, können Sie diese über die Schaltfläche "Ermitteln" lokalisieren (siehe folgende Abbildung).

Klicken Sie auf die *OK*-Schaltfläche, um die Referenz einzubinden und die entsprechenden Proxy-Klassen zu generieren.

HINWEIS: Prüfen Sie die *Refrences.cs*-Dateien, ob auch Proxy-Klassen mit den erforderlichen Datentypen vorhanden sind.



Konfigurieren der Silverlight-Oberfläche

Im nächsten Schritt erstellen wir unsere "umfangreiche" Silverlight-Test-Oberfläche (*MainPage.xaml*):

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    x:Class="Webdienst_Bsp.MainPage"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <sdk:DataGrid  AutoGenerateColumns="True" Name="dataGrid1" Background="AliceBlue" />
        <sdk:Label Height="50" HorizontalAlignment="Left" Margin="15,58,0,0"
            Name="label1" VerticalAlignment="Top" Content="Bitte warten ..." FontSize="22" />
    </Grid>
</UserControl>
```

Außer einem *Label*, das zur Geduld mahnt, fügen wir lediglich eine *DataGrid* ein. Vergessen Sie nicht, für das *DataGrid* die Eigenschaft *AutoGenerateColumns* auf *True* zu setzen.

Abrufen der Daten

Der eigentliche C#-Quellcode (*MainPage.xaml.cs*) ist schon etwas anspruchsvoller, haben doch Webdienste etc. in Silverlight den "Nachteil", dass sie nur asynchron abgerufen werden können. Eigentlich handelt es sich um keinen Nachteil, es soll lediglich sichergestellt werden, dass die Ausführung des Silverlight-Controls nicht sinnlos ausgebremst wird.

Sollten Sie also bereits mit der synchronen Einbindung von Webdiensten gearbeitet haben, müssen Sie jetzt etwas umdenken und etwas mehr Code produzieren:

```
...
namespace Daten1
{
    public partial class MainPage : UserControl
    {
...
```

Mit dem Laden des Controls setzt hektische Betriebsamkeit ein:

```
        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
```

Zunächst die Variante des WCF-Service:

Instanz des Proxys erzeugen:

```
            ServiceReference1.Service1Client mywebservice = new
                              ServiceReference1.Service1Client();
```

Eventhandler für den Datenabruf erstellen:

```
            mywebservice.GetAllProductsCompleted += new
                              EventHandler<ServiceReference1.GetAllProductsCompletedEventArgs>
                              (mywebservice_GetAllProductsCompleted);
```

Daten abrufen:

```
            mywebservice.GetAllProductsAsync();
```

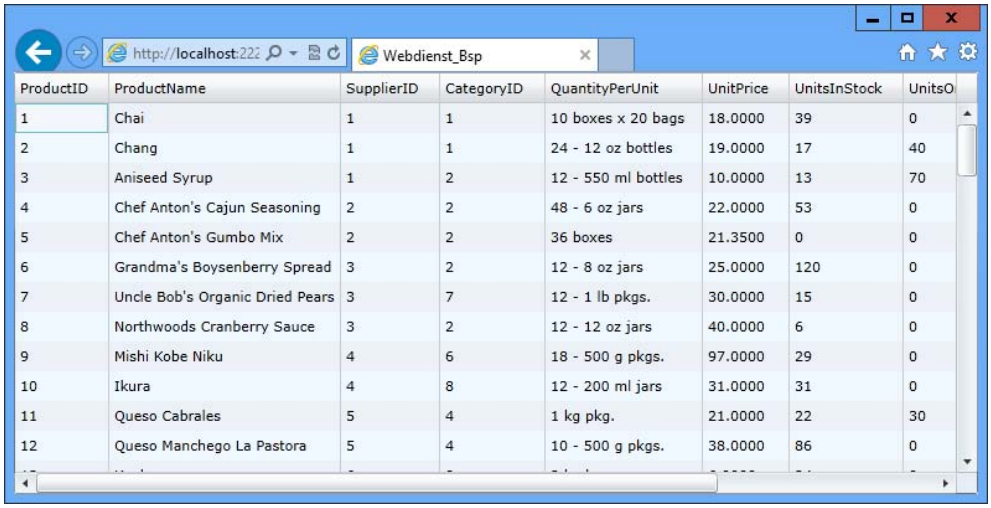
Alternativ der Webdienst mit der gleichen Vorgehensweise:

```
            ServiceReference2.WebService1SoapClient myws = new
                      ServiceReference2.WebService1SoapClient();
            myws.GetAllProductsCompleted += new
                 EventHandler<ServiceReference2.GetAllProductsCompletedEventArgs>(
                   myws_GetAllProductsCompleted);
            myws.GetAllProductsAsync();
        }
```

Hier die beiden Ereignishandler, die aufgerufen werden, wenn die Daten auf dem Client vorliegen:

```
        void myws_GetAllProductsCompleted(object sender,
                  ServiceReference2.GetAllProductsCompletedEventArgs e)
        {
            dataGrid1.ItemsSource = e.Result;
            label1.Visibility = System.Windows.Visibility.Collapsed;
        }

        void mywebservice_GetAllProductsCompleted(object sender,
                  ServiceReference1.GetAllProductsCompletedEventArgs e)
        {
           dataGrid1.ItemsSource = e.Result;
           label1.Visibility = System.Windows.Visibility.Collapsed;
        }
    }
}
```

Nach dem Start der Silverlight-Anwendung sollten jetzt – nach einigen Sekunden bangen Wartens – die gewünschten Daten im *DataGrid* erscheinen:



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsO
1	Chai	1	1	10 boxes x 20 bags	18.0000	39	0
2	Chang	1	1	24 - 12 oz bottles	19.0000	17	40
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.0000	13	70
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.0000	53	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.3500	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.0000	120	0
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.0000	15	0
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.0000	6	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.0000	29	0
10	Ikura	4	8	12 - 200 ml jars	31.0000	31	0
11	Queso Cabrales	5	4	1 kg pkg.	21.0000	22	30
12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.0000	86	0

Zusammenfassung

So weit – so gut, der Abruf von Tabellendaten als "Futter" für ListBoxen, DataGrids etc. dürfte nach obiger Anleitung kein Problem mehr darstellen. Das Zurückschreiben, Hinzufügen, Editieren und Löschen der Daten erfordert allerdings einen recht hohen Aufwand, auch hier müssen Sie für jede Operation eine entsprechende Methode erzeugen, die wiederum asynchron ausgeführt wird. Sie merken sicher, dass der Entwicklungsaufwand in diesem Fall schnell recht umfangreich werden kann.

Aus diesem Grund wurde, nicht zuletzt auch für Silverlight, das Konzept der WCF Data Services entwickelt, auf deren Verwendung wir im folgenden Abschnitt eingehen wollen

43.5.2 WCF Data Services

Auch in diesem Beispiel wollen wir ein bereits erstelltes Silverlight-Projekt mit dazugehörigem ASP.NET-Anwendungsprojekt als Ausgangspunkt verwenden.

Folgende Schritte werden wir abarbeiten:

- Einfügen einer SQL Server-Datenbank
- Erstellen eines Entity Data Models
- Erstellen des WCF Data Service
- Einbinden des WCF Data Service in das Silverlight-Projekt
- Konfigurieren der Silverlight-Oberfläche und Abrufen der Daten

Doch zunächst wollen wir kurz auf das Konzept der WCF Data Services eingehen.

Grundlagen

Bereits mit dem .NET 3.5 Service Pack 1 wurde ein weiteres Framework für das Erstellen von REST-basierten Datendiensten mitgeliefert, die "ADO.NET Data Services"[1]. Hintergrund dieser Technologie ist die Möglichkeit, auf einfache Weise komplexe Daten per REST-URI-Syntax freizugeben und mittels HTTP-Verben zu verarbeiten.

Grundlage ist ein per ADO.NET-Entity-Framework erzeugtes Datenmodell, das komplett oder in Teilen mit bestimmten Rechten freigegeben werden kann. Wer hier Parallelen zu den Webdiensten sieht, liegt nicht ganz falsch, der Ansatz ist in unserem Fall jedoch etwas komplexer, da hier die gesamte Datenstruktur im Mittelpunkt steht und kein SOAP-Protokoll eingesetzt wird.

Die vom Dienst zurückgegebenen XML-Daten lassen sich recht einfach mit den unterschiedlichsten Technologien auswerten und für die weitere Verarbeitung in den Clientanwendungen nutzen. Wer will kann sich also auf XML-Niveau herab begeben und die Datenoperationen auf diese Weise realisieren. Es geht jedoch auch einfacher, mittels Visual Studio können Sie auch hier Proxy-Klassen erzeugen, die den Zugriff auf die Daten wesentlich komfortabler gestalten.

Wie schon erwähnt, stellen die Data Services nicht einzelne Daten sondern ein Datenmodell zur Verfügung. Rufen Sie beispielsweise einen derartigen Service mit der Adresse

```
http://localhost:49654/WebDataService1.svc
```

ab, erhalten Sie die in der folgenden Abbildung gezeigte Antwort:

[1] Das war die alte Bezeichnung für diese Technologie.

So weit – so gut, doch was können wir damit anfangen? Eine kleine Änderung in der Adresszeile des Browsers zeigt uns beispielsweise die Inhalte der Tabelle *Artikel*:

```
http://localhost:49654/WebDataService1.svc/Artikel
```

Das Ergebnis dieser Anfrage ist folgender XML-Code:

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom"
xmlns:cf="http://www.microsoft.com/schemas/rss/core/2005"><channel
base="http://localhost:49654/WebDataService1.svc/"
xmlns:cfi="http://www.microsoft.com/schemas/rss/core/2005/internal"
cfi:lastdownloaderror="None"><title cf:type="text">Artikel</title><cf:guid
isPermaLink="false">http://localhost:49654/WebDataService1.svc/Artikel</cf:guid><atom:updated>20
08-07-10T10:40:13Z</atom:updated><lastBuildDate>Thu, 10 Jul 2008 10:40:13
GMT</lastBuildDate><atom:link href="Artikel" rel="self" title="Artikel"/>
<item><guid
isPermaLink="false">http://localhost:49654/WebDataService1.svc/Artikel(1)</guid><atom:updated
xmlns:atom="http://www.w3.org/2005/Atom">2008-07-10T10:40:13Z</atom:updated><atom:link
xmlns:atom="http://www.w3.org/2005/Atom" href="Artikel(1)" rel="edit"
title="Artikel"/><atom:link xmlns:atom="http://www.w3.org/2005/Atom"
href="Artikel(1)/Kategorien"
rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Kategorien"
type="application/atom+xml;type=entry" title="Kategorien"/><content type="application/xml"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
      <m:properties>
        <d:ArtikelNr m:type="Edm.Int32">1</d:ArtikelNr>
        <d:Artikelname>Chai</d:Artikelname>
        <d:LieferantenNr m:type="Edm.Int32">1</d:LieferantenNr>
        <d:Liefereinheit>10 Kartons x 20 Beutel</d:Liefereinheit>
        <d:Einzelpreis m:type="Edm.Decimal">18.0000</d:Einzelpreis>
        <d:Lagerbestand m:type="Edm.Int16">18</d:Lagerbestand>
        <d:BestellteEinheiten m:type="Edm.Int16">0</d:BestellteEinheiten>
        <d:Mindestbestand m:type="Edm.Int16">10</d:Mindestbestand>
        <d:Auslaufartikel m:type="Edm.Boolean">false</d:Auslaufartikel>
      </m:properties>
```

```
    </content>
...
      <m:properties>
        <d:ArtikelNr m:type="Edm.Int32">2</d:ArtikelNr>
        <d:Artikelname>Chang</d:Artikelname>
        <d:LieferantenNr m:type="Edm.Int32">1</d:LieferantenNr>
        <d:Liefereinheit>24 x 12-oz-Flaschen</d:Liefereinheit>
        <d:Einzelpreis m:type="Edm.Decimal">19.0000</d:Einzelpreis>
        <d:Lagerbestand m:type="Edm.Int16">17</d:Lagerbestand>
        <d:BestellteEinheiten m:type="Edm.Int16">40</d:BestellteEinheiten>
        <d:Mindestbestand m:type="Edm.Int16">25</d:Mindestbestand>
        <d:Auslaufartikel m:type="Edm.Boolean">false</d:Auslaufartikel>
      </m:properties>
```

Zurückgegeben wird eine gut strukturierte und typisierte Liste der Tabelle *Artikel* unserer Datenbank. Doch es geht auch noch detaillierter, wenn Sie zum Beispiel direkt auf einen spezifischen Artikel zugreifen wollen. In diesem Fall geben Sie einfach zusätzlich den Index bei der Abfrage an:

```
http://localhost:49654/WebDataService1.svc/Artikel(3)
```

Der Rückgabewert ist das entsprechende Objekt in der Collection:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xml:base="http://localhost:49654/WebDataService1.svc/"
xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
m:type="testModel.Artikel" xmlns="http://www.w3.org/2005/Atom">
  <id>http://localhost:49654/WebDataService1.svc/Artikel(3)</id>
  <title type="text"></title>
  <updated>2008-07-10T11:40:35Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="Artikel" href="Artikel(3)" />
  <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Kategorien"
type="application/atom+xml;type=entry" title="Kategorien" href="Artikel(3)/Kategorien" />
  <content type="application/xml">
    <m:properties>
      <d:ArtikelNr m:type="Edm.Int32">3</d:ArtikelNr>
      <d:Artikelname>Aniseed Syrup</d:Artikelname>
      <d:LieferantenNr m:type="Edm.Int32">1</d:LieferantenNr>
      <d:Liefereinheit>12 x 550-ml-Flaschen</d:Liefereinheit>
      <d:Einzelpreis m:type="Edm.Decimal">112.0000</d:Einzelpreis>
      <d:Lagerbestand m:type="Edm.Int16">13</d:Lagerbestand>
      <d:BestellteEinheiten m:type="Edm.Int16">70</d:BestellteEinheiten>
      <d:Mindestbestand m:type="Edm.Int16">25</d:Mindestbestand>
      <d:Auslaufartikel m:type="Edm.Boolean">false</d:Auslaufartikel>
    </m:properties>
  </content>
</entry>
```

Unsere Silverlight-Applikation kann jedoch auch über die von Visual Studio erzeugten Proxy-Klassen und damit über ein *DataServiceContext*-Objekt auf die veröffentlichten Daten zugreifen. Auf dem Client können Sie also mit LINQ und typisierten Klassen wie gewohnt arbeiten (Daten anzeigen, löschen, editieren). Doch der Reihe nach ...

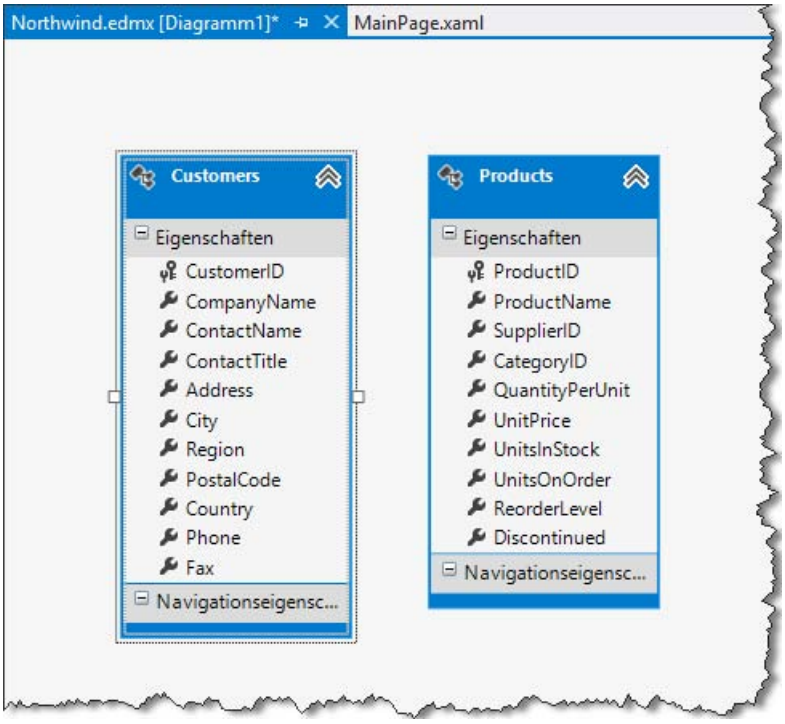
Einfügen der SQL Server-Datenbank

Auch in diesem Fall erstellen wir keine neue Datenbank (Einbindung von *Northwind.mdf* in das Projekt siehe Seite 2139).

Erstellen des Entity Data Models

Im Gegensatz zu unserem letzten Beispiel benötigen wir jetzt in jedem Fall ein ADO.NET Entity DataModel, da nur dieses per *WCF Data Service* freigegeben werden kann. Erstellen Sie also mittels Projektmappen Explorer (Kontextmenü *Hinzufügen|Neues Element hinzufügen*) ein ADO.NET Entity DataModel.

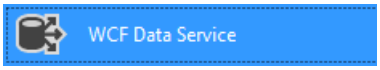
Wählen Sie im folgenden Assistenten als Datenquelle unsere *Northwind*-Datenbank aus und fügen Sie die beiden Tabellen *Customers* und *Products* in das Datenmodell ein:



Damit ist zunächst das Datenmodell mit zwei Entitäten erzeugt und wir können uns der Web-Schnittstelle zuwenden.

Erstellen des WCF Data Service

Fügen Sie über den Projektmappen Explorer dem ASP.NET-Testprojekt (Kontextmenü *Hinzufügen|Neues Element hinzufügen*) einen WCF Data-Service hinzu.



Öffnen Sie nachfolgend die Datei *WcfDataService1.svc.cs* und nehmen Sie folgende Änderungen vor:

Wir "zapfen" das gewünschte Entity-Modell an:

```
public class WcfDataService1 : DataService<NorthwindEntities>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
```

Hier bestimmen wir, welche Entitäten wir freigeben und welche Zugriffsrechte erteilt werden:

```
        config.SetEntitySetAccessRule("Customers", EntitySetRights.All);
        config.SetEntitySetAccessRule("Products", EntitySetRights.All);
        config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
    }
}
}
```

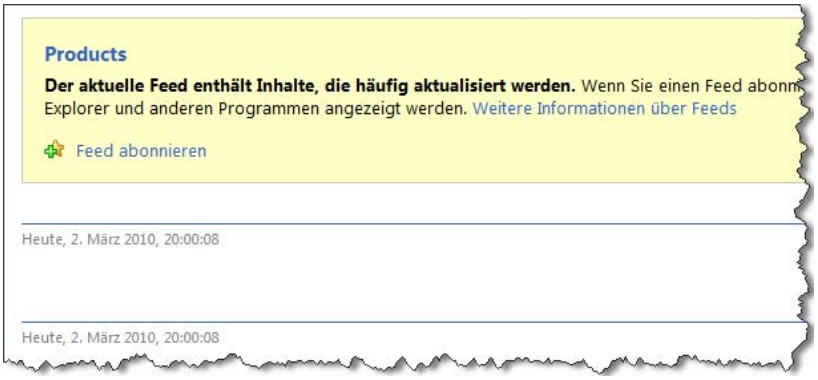
HINWEIS: Im Normalfall sollten Sie mit den erteilten Zugriffsrechten nicht ganz so sorglos umgehen, mit obiger Einstellung ist auch das Löschen von Daten kein Problem, und wie Sie gesehen haben, braucht es nicht einmal einen extra Client um mit den Daten arbeiten zu können, der Webbrowser genügt vollkommen.

Dies ist auch gleich das Stichwort: Starten Sie ruhig einmal das Projekt und geben Sie folgende Parameter im Browser an:

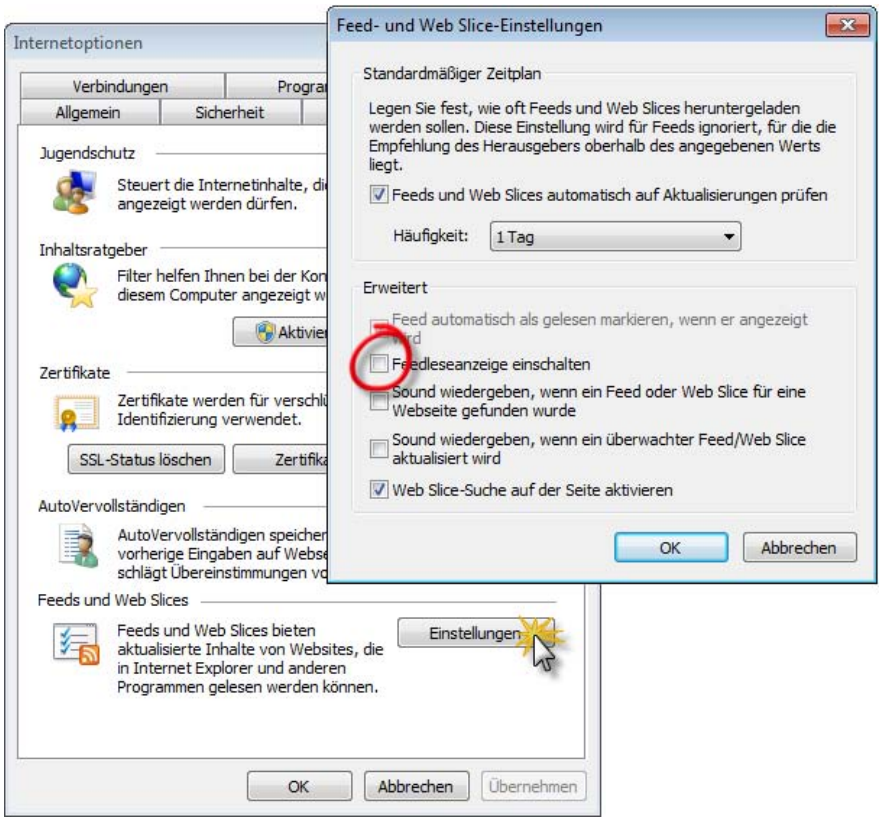
```
http://localhost:27696/WcfDataService1.svc/Products
```

Sie sollten jetzt eine XML-Ansicht der Artikel aus unserer Datenbank erhalten:

Eventuell kann Ihnen der Browser auch eine Feed-Anzeige präsentieren, die uns allerdings hier nicht weiterhilft:



Um diese Ansicht auszuschalten, wechseln Sie in die Internetoptionen und entfernen Sie das Häkchen bei "Feedleseanzeige einschalten":



Spätestens jetzt sollte auch eine Anzeige der XML-Daten erfolgen.

Einbinden des WCF Data Service in das Silverlight-Projekt

Damit der Client (Silverlight) mit den Informationen des WCF Data Service auch etwas anfangen kann, müssen wir einen Webverweis hinzufügen und damit auch entsprechende Proxy-Klassen erzeugen. Wählen Sie also das Silverlight-Projekt aus und fügen Sie eine Referenz auf den gerade erstellten WCF Data Service hinzu.

Konfigurieren der Silverlight-Oberfläche

Wie schon beim letzten Beispiel wollen wir es bei einer schlichten Oberfläche belassen, die lediglich aus einem *DataGrid* besteht. Beachten Sie auch die zwei Ereignishandler:

```
<UserControl x:Class="Daten.MainPage"
...
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400" Loaded="UserControl_Loaded_1">
    <StackPanel>
```

```
        <sdk:DataGrid AutoGenerateColumns="True"  Name="dataGrid1"
                      RowEditEnded="dataGrid1_RowEditEnded" />
    </StackPanel>
</UserControl>
```

Abrufen der Daten

Und damit sind wir auch hier fast schon am Ende angelangt. Wie auch bei den Webdiensten müssen Sie bei Verwendung der WCF Data Services darauf achten, dass Sie diese asynchron abrufen, der synchrone Zugriff wird von Silverlight nicht unterstützt.

Öffnen Sie also die Datei *MainPage.xaml.cs* und nehmen Sie folgende Erweiterungen vor:

```
...
using System.Data.Services.Client;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

namespace Daten
{
    public partial class MainPage : UserControl
    {
```

Eine Instanz für die Proxy-Klasse:

```
        ServiceReference1.NorthwindEntities svcContext;
...
```

Mit dem Laden der Seite beginnen die Aktivitäten:

```
        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
```

DataContext mit Verweis auf den Datendienst erzeugen:

```
            svcContext = new ServiceReference1.NorthwindEntities(
                             new Uri("WcfDataService1.svc", UriKind.Relative));
```

Da ein synchroner Zugriff nicht möglich ist, erzeugen wir eine *DataServiceQuery*:

```
            DataServiceQuery<ServiceReference1.Products> query = svcContext.Products;
```

Diese wird asynchron ausgeführt:

```
            query.BeginExecute(new AsyncCallback(r =>
                {
```

Zuweisen der Rückgabedaten:

```
                    dataGrid1.ItemsSource = query.EndExecute(r).ToList();
                }), null);
        }
...
```

Wie Sie sehen verwenden wir einen Lambda-Ausdruck für das Ende der asynchronen Operation, mit dem wir den Rückgabewert von *EndExecute* dem *DataGrid* als *ItemsSource* zuweisen.

Damit steht einem Test der Anwendung nichts mehr im Weg. Auch hier sollte nach kurzer Wartezeit (etwas mehr als bei den Webdiensten) ein gefülltes *DataGrid* angezeigt werden.

Bearbeiten der Daten

Doch wie sieht es eigentlich mit dem Bearbeiten der Daten aus? Auch das ist jetzt kein Problem mehr. Statt am Server, d h. am WCF Data Service, herumzudoktern und endlose Erweiterungen vorzunehmen genügt es, wenn wir den Client entsprechend erweitern, der Datendienst verfügt bereits über die nötige Funktionalität.

Wir hatten ja bereits eine entsprechende Ereignisprozedur (*RowEditEnded*) für das *DataGrid* vorgesehen, kommen wir also jetzt zur Implementierung:

```
        private void dataGrid1_RowEditEnded(object sender, DataGridRowEditEndedEventArgs e)
        {
```

Zunächst benötigen wir den bearbeiteten Artikel, d h. die *Products*-Instanz der bearbeiteten Zeile:

```
            ServiceReference1.Products p = (dataGrid1.ItemsSource as
                           List<ServiceReference1.Products>)[e.Row.GetIndex()];
```

HINWEIS: Mit Bezug auf die Ereignisparameter können Sie obiges schlecht lesbares Konstrukt verwenden, es geht aber alternativ auch so:

```
            ServiceReference1.Products p = (dataGrid1.SelectedItem as
                                            ServiceReference1.Products);
```

Wir aktualisieren das Objekt zunächst noch lokal:

```
            svcContext.UpdateObject(p);
```

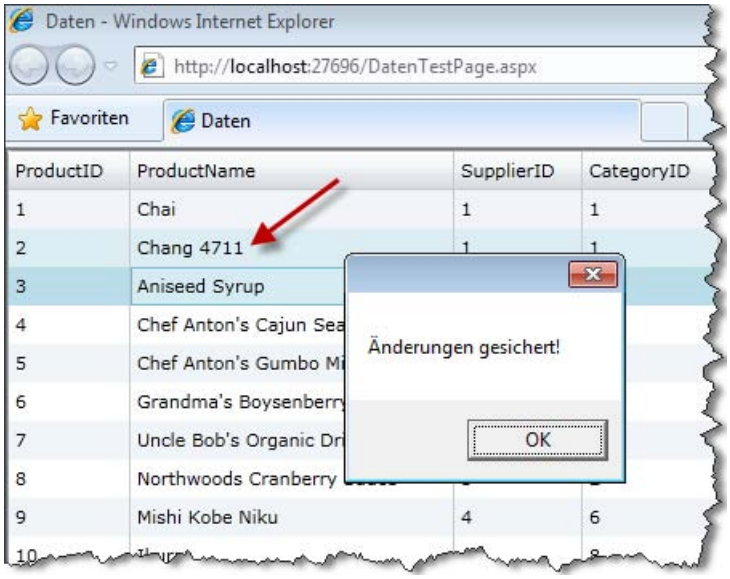
Hier schreiben wir die Änderungen an den Server zurück, über das Ergebnis werden wir asynchron unterrichtet:

```
            svcContext.BeginSaveChanges(new AsyncCallback(r =>
                {
```

Eine kurze Meldung sollte genügen:

```
                    MessageBox.Show("Änderungen gesichert!");
                }), true);
        }
```

Ein Test wird Sie von der Funktionsfähigkeit überzeugen, ändern Sie einfach einen Produktnamen und warten Sie auf die Vollzugsmeldung per MessageBox.



Auf ähnliche Art und Weise lassen sich auch die anderen CRUD-Operationen realisieren, wir wollen an dieser Stelle nicht weiter darauf eingehen.

Arbeiten mit LINQ und Erweiterungsmethoden

Wer jetzt glaubt, dass damit alle Möglichkeiten der Data Services ausgeschöpft sind, dürfte sich täuschen. Auf dem Client müssen Sie nicht mit "dummen" Listen arbeiten, Sie können sich auch im fast unerschöpflichen Fundus von LINQ bedienen.

BEISPIEL 43.17: Verwendung von LINQ-Abfragen auf dem Client

C#

```
...
  svcContext = new ServiceReference1.NorthwindEntities(new Uri("WcfDataService1.svc",
                                                        UriKind.Relative));
```

Hier steht die LINQ-Abfrage:

```
  DataServiceQuery<ServiceReference1.Products> query =
            (DataServiceQuery<ServiceReference1.Products>) svcContext.Products.Take(5);
```

HINWEIS: Eine Typisierung des Abfrageergebnisses ist zwingend notwendig!

Alternativ können Sie zum Beispiel auch alle Artikel abfragen, die mit "A" beginnen:

```
 DataServiceQuery<ServiceReference1.Products> query =
       (DataServiceQuery<ServiceReference1.Products>) svcContext.Products.Where(p =>
                                                      p.ProductName.StartsWith("A"));
```

HINWEIS: Um Missverständnissen vorzugreifen: Bei den oben definierten Abfragen handelt es sich um die Vorlagen für die finalen SQL-Abfragen, d.h., im ersten Fall werden auch bloß 5 Elemente vom Server an den Client gesendet.

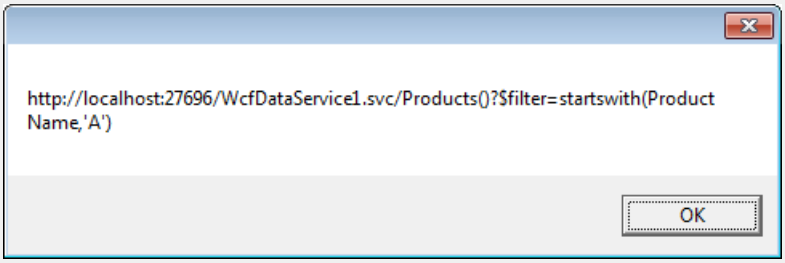
Mit einer einfachen Programmerweiterung können Sie auch die Uri-Befehle, die an den Server gesendet werden, ermitteln:

BEISPIEL 43.18: Anzeige der Abfragezeichenkette

C#

```
...
   DataServiceQuery<ServiceReference1.Products> query =
             (DataServiceQuery<ServiceReference1.Products>) svcContext.Products.Where(p =>
                  p.ProductName.StartsWith("A"));
   MessageBox.Show(query.RequestUri.ToString());
...
```

Ergebnis



Damit dürfte auch schnell die Leistungsfähigkeit der WCF-Data Services klar werden. Doch Vorsicht:

HINWEIS: Haben Sie die Daten einmal abgerufen und wenden Sie auf diese Ergebnisse LINQ-Abfragen an, werden diese lediglich auf dem Client ausgeführt.

An dieser Stelle könnten wir sicher noch recht tief in die Thematik "Data Services" oder auch "RIA-Services"[1] eintauchen, aber dazu fehlt uns hier einfach der Platz. Mehr zu diesen Themen finden Sie in unseren Datenbankbüchern.

Stattdessen wollen wir uns noch einem Spezialthema widmen, das im direkten Zusammenhang mit der Funktion des Silverlight-Plug-in steht.

[1] Rich Internet Application-Services, der noch komplexere Nachfolger für die WCF Data Services, mit integrierter Validierung und Authentifizierung etc.

43.6 Isolierter Speicher

Jeder Webprogrammierer steht früher oder später vor der Aufgabe, Informationen auf dem Client abzulegen. Ein direkter Zugriff auf das Dateisystem des Client-PCs ist aber aus Sicherheitsgründen ausgeschlossen, was bereits eine wesentliche Einschränkung darstellt.

Hier fällt manchem sicher das Konzept der Cookies ein, kleine Dateien, die in einem speziellen Verzeichnis des Browsers abgelegt werden können und beim nächsten Zugriff der Anwendung wieder verfügbar sind.

43.6.1 Grundkonzept

Mit Silverlight wurde ein wesentlich leistungsfähigeres Konzept namens *Isolated Storage* eingeführt, das weit über die Fähigkeiten von Cookies hinausgeht. So sind Sie zum einen nicht mehr auf Gedeih und Verderb an den Browser-Cache gebunden, in dem die Cookies abgelegt werden (und evtl. auch gelöscht), auch die Größenbeschränkung auf 4 Kilobyte ist nicht mehr relevant. Statt dessen arbeiten Sie sehr komfortabel mit einem eigenen "Dateisystem", das Pfade und Dateien enthalten kann.

Diese Funktionalität wird durch den Namespace *System.IO.IsolatedStorage* bereitgestellt, für den Zugriff nutzen Sie die *IsolatedStorageFile*-Klasse, die unter anderem die statischen Methoden *GetUserStoreForApplication* und *GetUserStoreForSite* bereitstellt.

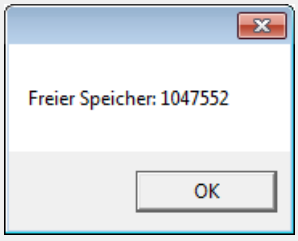
Während *GetUserStoreForApplication* isolierten Speicher für eine spezifische Anwendung bereitstellt (das ist Ihre Silverlight-Anwendung), bietet *GetUserStoreForSite* isolierten Speicher für einen anwendungsübergreifenden Zugriff.

BEISPIEL 43.19: Abrufen des freien Speichers

C#

```
...
using System.IO.IsolatedStorage;
...
    private void button1_Click(object sender, RoutedEventArgs e)
       {
           IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
           MessageBox.Show("Freier Speicher: " + isof.AvailableFreeSpace.ToString());
       }
```

Ergebnis



Aufbauend auf dem zurückgegebenen *IsolatedStorageFile*-Objekt können Sie das virtuelle Dateisystem verwalten, d h., Sie können Dateien und Verzeichnisse lesen und schreiben sowie Informationen über das Dateisystem abfragen.

43.6.2 Das virtuelle Dateisystem verwalten

Wie bereits das vorhergehende Beispiel gezeigt hat, bietet die *IsolatedStorageFile*-Klasse die Möglichkeit, auf das virtuelle Dateisystem des isolierten Speichers zuzugreifen. Ein Blick auf die verfügbare Größe dürfte bereits zeigen, dass es sich hier um einen beschränkten Bereich auf dem Client-System handelt, der sich je nach System in einem anderen Verzeichnis befindet:

System	Ablagepfad
Windows XP	`C:\Dokumente und Einstellungen\<Anwendername>\Anwendungsdaten\Microsoft\Silverlight\is`
Windows Vista Windows 7 Windows 8	`C:\Users\<Anwendername>\AppData\LocalLow\Microsoft\Silverlight\is`
Mac OS X	`AppData\Local`

Erzeugen Sie für Ihre Anwendung einen neuen isolierten Speicher, wird in den oben angegebenen Verzeichnissen ein neues Unterverzeichnis erstellt.

HINWEIS: Der Name des Verzeichnisses ist verschlüsselt, der Inhalt jedoch nicht. Deshalb sollten Sie auch keine Passwörter, Connectionstrings etc. in diesem Speicher ablegen, ohne sie vorher selbst zu verschlüsseln.

Beachten Sie auch, dass der isolierte Speicher sowohl an an die Anwendung (oder alternativ an die gesamte Site) als auch auch an den aktuellen Nutzer gebunden ist, was aus den obigen Ablageverzeichnissen eindeutig hervorgeht.

Wieviel Platz ist verfügbar?

Hier müssen wir zwischen zwei Werten unterscheiden – der prinzipiell verfügbare und der noch freie Platz. Für diese Informationen stehen Ihnen die Eigenschaften *Quota* (Gesamtplatz) und *AvailableFreeSpace* (freier Speicher) zur Verfügung.

BEISPIEL 43.20: Abfrage Gesamtspeicher und verfügbarer Speicher

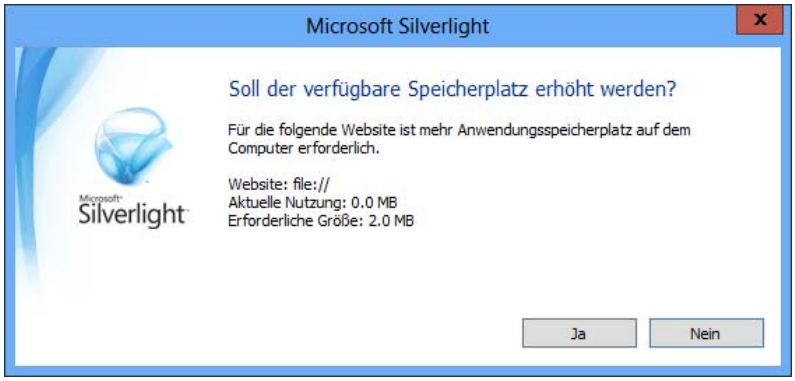
C#

```
IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
MessageBox.Show("Gesamt Speicher: " + isof.Quota.ToString());
MessageBox.Show("Freier Speicher: " + isof.AvailableFreeSpace.ToString());
```

HINWEIS: Standardmäßig ist *Quota* auf 1048576, d h. 1 MByte, festgelegt, die merkwürdige Zahl ergibt sich aus der Berechnung eines MBytes.

Ich will mehr!

Mancher bekommt nie genug und so werden sicher auch die 1 MByte nicht lange reichen und schnell zu dem Wunsch führen, etwas mehr zu bekommen. Auch das ist prinzipiell kein Problem, mit der *IncreaseQuotaTo*-Methode können Sie weiteren Speicher anfordern. "Anfordern" ist auch das richtige Wort dafür, da der Nutzer per Dialog darüber entscheidet, ob Sie das gewünschte Kontingent auch erhalten:



Unterverzeichnisse erzeugen

Das Erzeugen von Unterverzeichnissen dient der besseren Übersicht im isolierten Speicher und ist mit der intuitiven Methode *CreateDirectory* realisierbar. Sie müssen lediglich den Namen des neuen Ordners übergeben.

HINWEIS: Ist das Verzeichnis bereits vorhanden wird **kein** Fehler ausgelöst!

BEISPIEL 43.21: Erzeugen von Unterverzeichnissen

C#
```
IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
isof.CreateDirectory("BilderCache/Klein");
isof.CreateDirectory("BilderCache/Gross");
```

HINWEIS: Sie können auch mehrere Verzeichnisse gleichzeitig erzeugen, siehe obiges Beispiel.

Anzeige von Dateien und Verzeichnissen

Verzeichnisse im isolierten Speicher können Sie mit der Methode *GetDirectoryNames* bestimmen, Dateien ermitteln Sie mit der Methode *GetFileNames*.

BEISPIEL 43.22: Ermitteln aller Dateien und Verzeichnisse in der Root des isolierten Speichers

C#
```
IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
string[] dirs = isof.GetDirectoryNames("*");
foreach (string s in dirs)
```

BEISPIEL 43.22: Ermitteln aller Dateien und Verzeichnisse in der Root des isolierten Speichers

```
            MessageBox.Show(s);
        string[] files = isof.GetFileNames("*");
        foreach (string s in files)
            MessageBox.Show(s);
```

Neben dem oben gezeigten Platzhalterzeichen können Sie auch das "?" für ein beliebiges Zeichen verwenden.

Dateien/Verzeichnisse löschen

Mit den Methoden *DeleteDirectory* bzw. *DeleteFile* können Sie Verzeichnisse bzw. Dateien auch wieder löschen. Die Verwendung dürfte nicht allzu anspruchsvoll sein:

BEISPIEL 43.23: Verzeichnis löschen

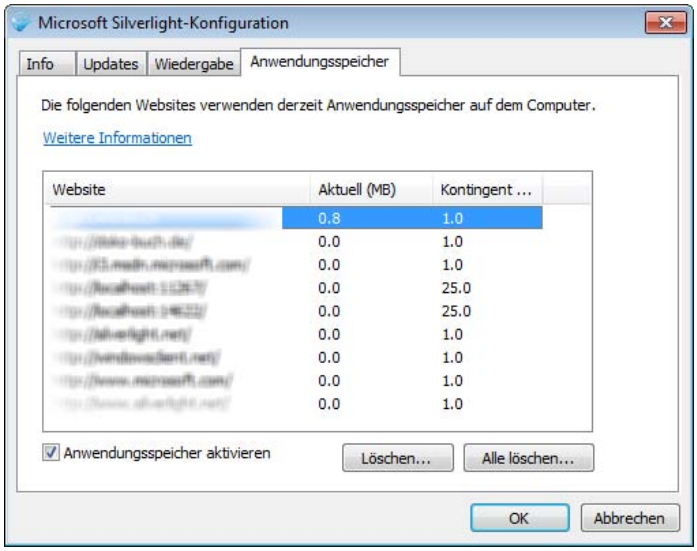
C#

```
        IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
        isof.DeleteDirectory("BilderCache/Klein");
```

HINWEIS: Ist das Verzeichnis nicht vorhanden, wird ein Fehler ausgelöst, gleiches trifft zu, wenn das zu löschende Verzeichnis nicht leer ist.

Sie müssen sich also in jedem Fall die Mühe machen, gegebenenfalls sogar rekursiv das Verzeichnis zu durchforsten, um alle Dateien und Unterverzeichnisse zu löschen.

Über das Kontextmenü des Silverlight-Plug-in können Sie auch als Endanwender recht bequem den isolierten Speicher einer Anwendung/Site löschen:



43.6.3 Arbeiten mit Dateien

Nachdem wir uns bisher nur mit administrativen Aufgaben im isolierten Speicher beschäftigt haben, wollen wir jetzt endlich auch zum eigentlichen Sinn des Ganzen kommen: das Schreiben und Lesen von Dateien.

Eine Datei können Sie mit Hilfe der *IsolatedStorageFileStream*-Klasse erzeugen, bearbeiten und später auch lesen.

Schreiben

Ist die *IsolatedStorageFileStream*-Instanz erzeugt, können Sie Inhalte mit Hilfe eines *StreamWriters* hinzufügen.

BEISPIEL 43.24: Erzeugen einer neuen Datei

C#
```
IsolatedStorageFile isof = IsolatedStorageFile.GetUserStoreForApplication();
isof.CreateDirectory("BilderCache/Gross");
IsolatedStorageFileStream istream = new
                IsolatedStorageFileStream("/BilderCache/Gross/Index.txt",
                                          FileMode.Create, isof);
StreamWriter writer = new StreamWriter(istream);
for (int i = 0 ; i < 100; i++)
   writer.WriteLine("Zeile " + i.ToString());
writer.Close();
istream.Close();
```

Welche Möglichkeiten beim Öffnen der Datei bestehen, zeigt die folgende Tabelle:

FileMode	Beschreibung
Append	Datei zum Anhängen von Inhalten öffnen.
Create	Erzeugt eine neue, leere Datei mit dem gewünschten Namen, ist die Datei vorhanden, wird sie überschrieben.
CreateNew	Erzeugt eine neue Datei, ist diese bereits vorhanden, wird eine Fehler ausgelöst.
Open	Öffnet eine bestehende Datei.
OpenOrCreate	Öffnet eine bestehende Datei, ist diese nicht vorhanden, wird sie erzeugt.
Truncate	Öffnet eine bestehende Datei und löscht deren Inhalte.

Lesen von Dateien

Auch hier erstellen Sie zunächst eine *IsolatedStorageFileStream*-Instanz (z.B. mit *FileMode.Open*) und nutzen dann einen *StreamReader.* um die Daten einzulesen.

BEISPIEL 43.25: Einlesen von Daten und Anzeige ein einer *ListBox*

C#
```
IsolatedStorageFileStream istream = new
          IsolatedStorageFileStream("/BilderCache/Gross/Index.txt",
```

BEISPIEL 43.25: Einlesen von Daten und Anzeige ein einer *ListBox*

```
                                        FileMode.Open, isof);
StreamReader reader  = new StreamReader(istream);
string zeile = "";
while ((zeile = reader.ReadLine()) != null)
    Listbox1.Items.Add(zeile);
reader.Close();
istream.Close();
```

HINWEIS: Wie Sie mit *StreamReader* bzw. *StreamWriter* arbeiten können, soll nicht Gegenstand dieses Kapitel sein.

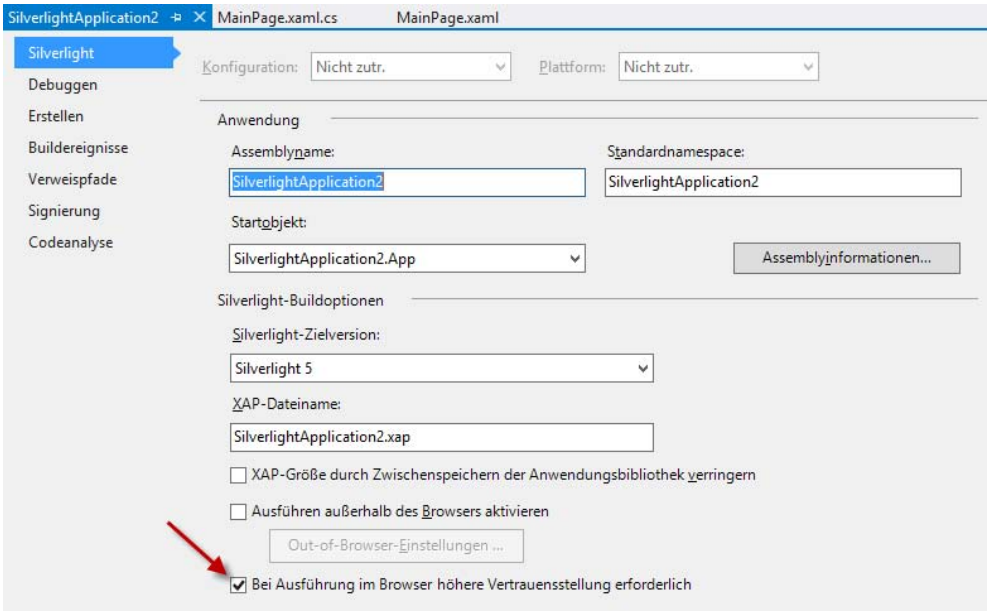
43.7 Fulltrust-Anwendungen

Mehrfach war bereits die Rede von einer FullTrust-Anwendung oder auch von höherer Vertrauensstellung. Beides besagt, dass die Silverlight-Anwendung unter bestimmten Umständen auf Ressourcen zugreifen darf, die sonst nicht verfügbar sind.

Grundsätzlich müssen Sie in diesem Zusammenhang von zwei Szenarien ausgehen:

- Ausführung im Browser
- Ausführung als Out-of-Browser-Anwendung

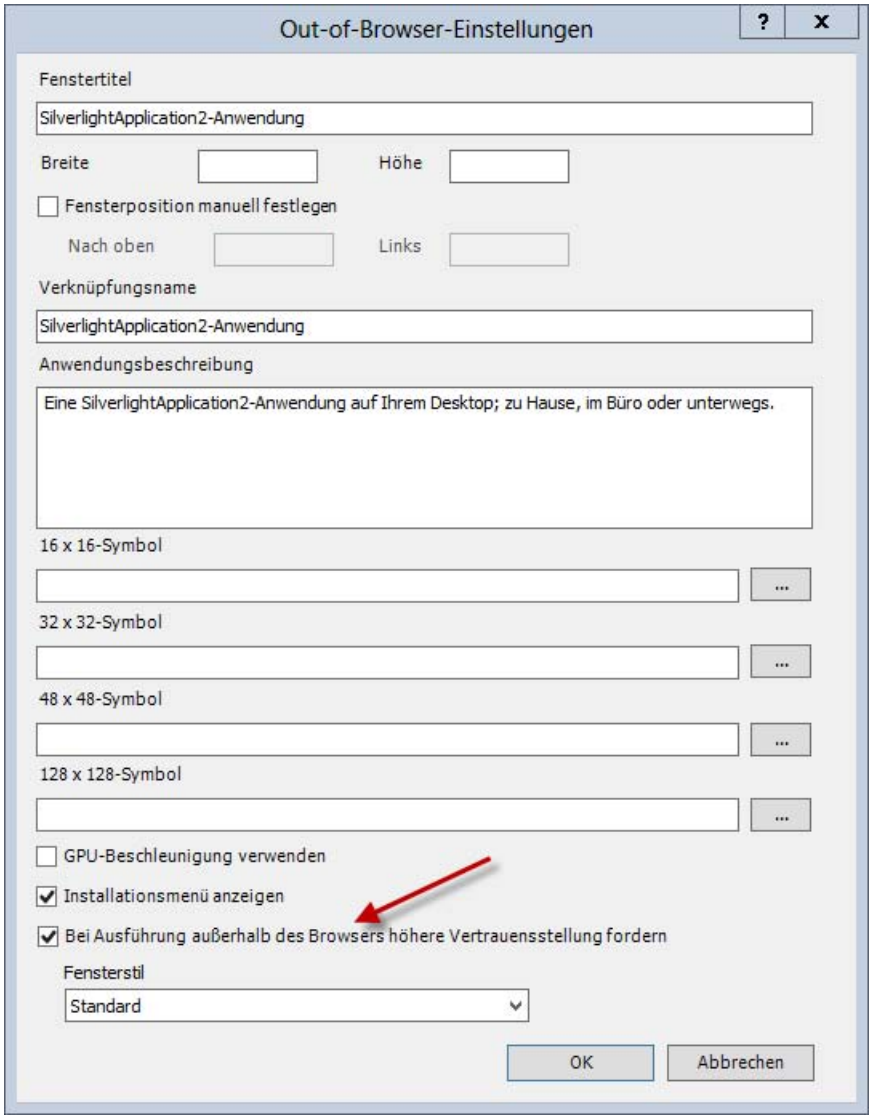
Möchten Sie im ersten Fall eine höhere Vertrauensstellung erreichen, können Sie diese zunächst über die Projektoptionen anfordern:



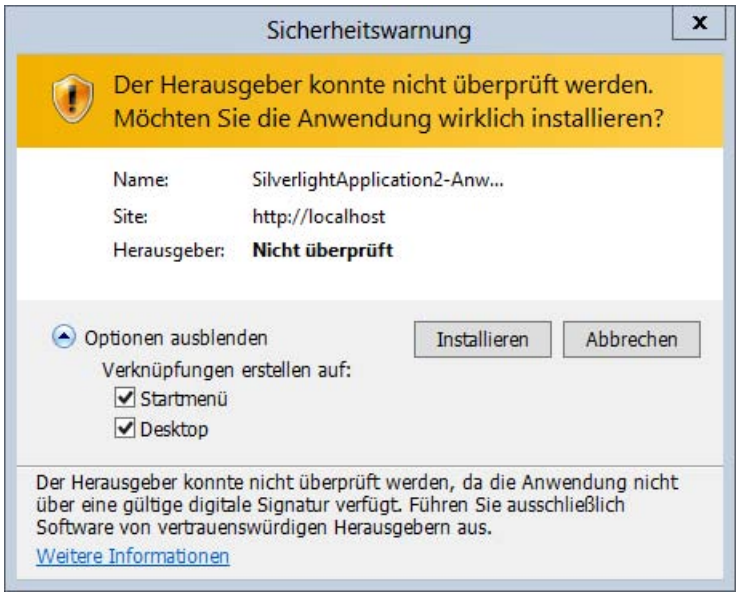
Das Häkchen hier besagt aber nicht, dass Ihre Anwendung diese Stellung auch erhält. Zwei Voraussetzungen sind bei der Browser-Variante erforderlich:

- Die Anwendung ist digital mit einem vertrauenswürdigen Zertifikat signiert.
- Die Erhöhung der Vertrauensstellung ist per Client Access Policy-Datei oder per Registry-Änderung freigegeben.

Variante 2, d.h. die Ausführung als Out-Of-Browser-Anwendung, ist da schon etwas einfacher realisierbar. Hier genügt es zunächst, in den Out-of-Browser-Einstellungen die Option *Bei Ausführung außerhalb des Browsers höhere Vertrauensstellung fordern* zu aktivieren:



Versucht jetzt der Anwender eine derartige Silverlight-Anwendung lokal zu installieren, wird zunächst eine Sicherheitsmeldung eingeblendet:



Wird die Meldung durch den Nutzer bestätigt, kann Ihre Anwendung jetzt von den erhöhten Rechten profitieren und zum Beispiel direkt auf das Dateisystem des Anwenders zugreifen:

BEISPIEL 43.26: Zugriff auf das Dateisystem

C#

```
...
using System.IO;
...
       private void Button_Click_1(object sender, RoutedEventArgs e)
       {
```

Handelt es sich um eine Fulltrust-Anwendung?

```
           if (Application.Current.HasElevatedPermissions)
           {
```

Wenn ja, erzeugen wir eine neue Datei auf dem Desktop des Anwenders:

```
               string path = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
               File.WriteAllText(path + @"\test.txt", "Hallo World!!!!!!");
           }
           else
              MessageBox.Show("Leider reichen die Rechte nicht!");
       }
...
```

Weitere Beispiele finden Sie im folgenden Abschnitt.

43.8 Praxisbeispiele

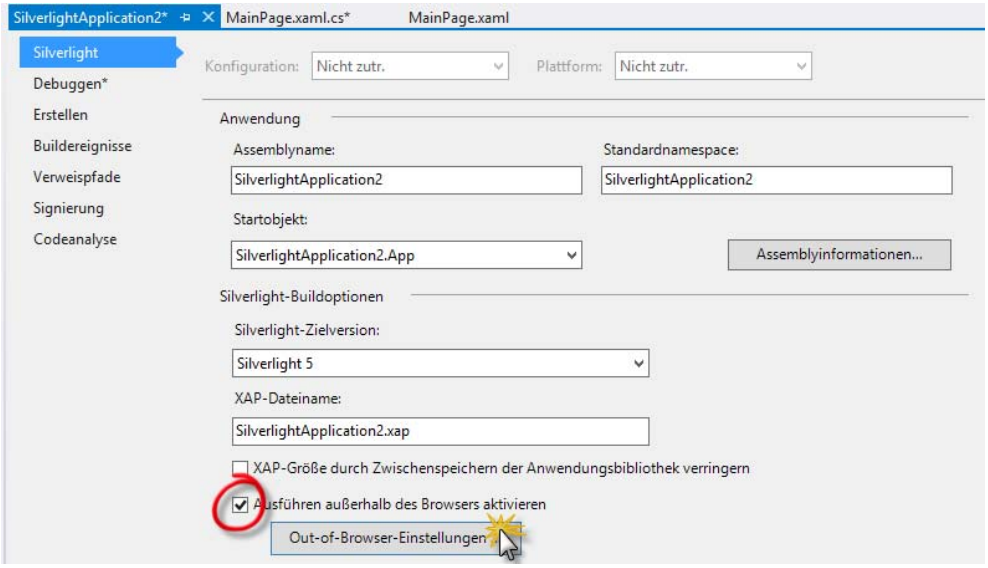
43.8.1 Eine Out-of-Browser-Applikation realisieren

Wollen Sie den ClickOnce-WPF-Anwendungen Konkurrenz machen? Wenn ja, bietet Ihnen auch Silverlight die Möglichkeit, die Silverlight-Applikationen als Out-of-Browser-Anwendung direkt im Startmenü und auf dem Desktop des Endanwenders zu installieren. Eine Online-Verbindung ist für die Anwendung nur dann nötig, wenn Sie beispielsweise auf Webdienste etc. zugreifen wollen.

Ausgehend von einer simplen Silverlight-Anwendung (z.B. ein *Calendar*-Control und eine zusätzliche Schaltfläche) wollen wir Ihnen zeigen, wie Sie Ihre Anwendung konfigurieren können/müssen, damit sie auch out-of-Browser funktioniert.

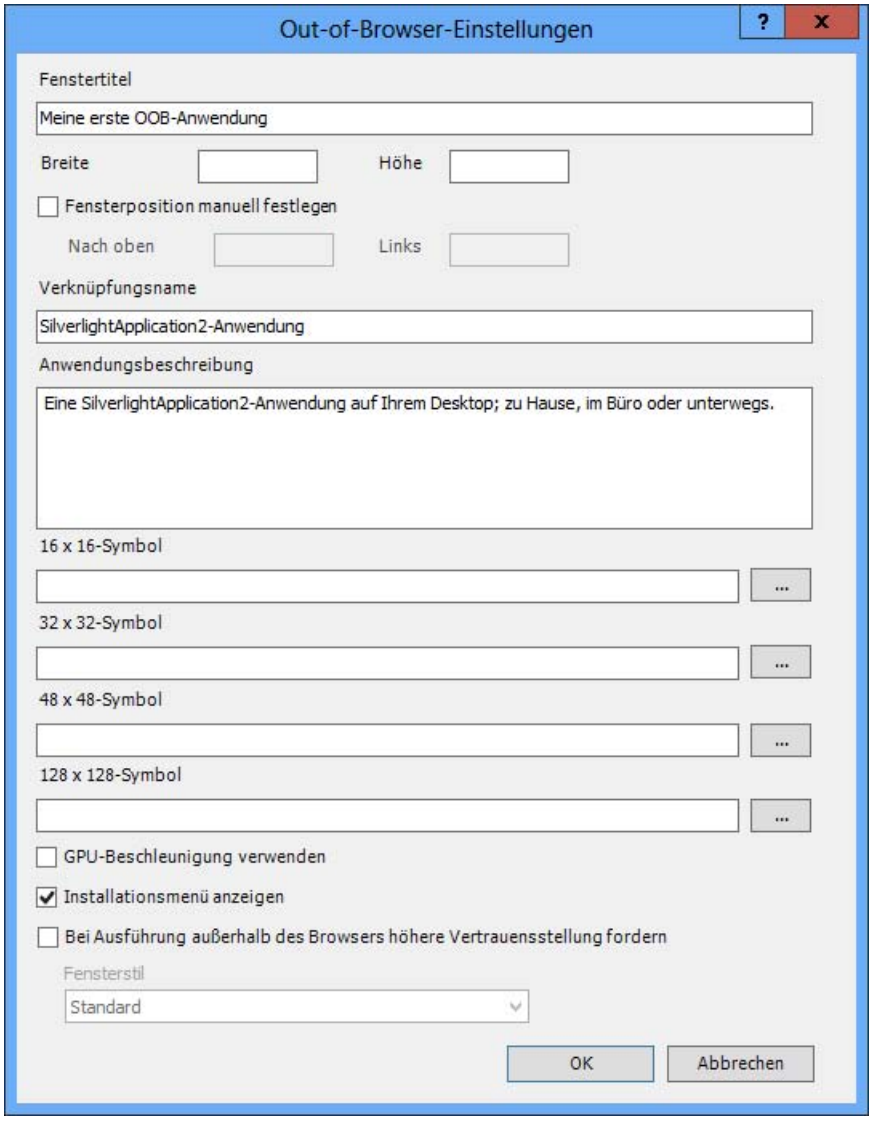
Vorgehensweise

Nach dem Erstellen des Projekts wechseln Sie bitte in die Projekteigenschaften und markieren die Option "Ausführen der Anwendung außerhalb des Browsers aktivieren":



Nachfolgend ist die Anwendung bereits so konfiguriert, dass ein externen Betrieb möglich ist (die Datei *AppManifest.xaml* wurde angepasst). Es ist jedoch ratsam, sich auch mit den Einzeloptionen für den Out-Of-Browser-Betrieb vertraut zu machen. Rufen Sie diese über die entsprechende Schaltfläche im obigen Dialog auf.

HINWEIS: Alle hier vorgenommenen Einstellungen wirken sich direkt auf die finale *AppManifest.xaml*-Datei aus.



Die Optionen in obigem Konfigurationsdialog sollten selbsterklärend sein.

BEISPIEL 43.27: Die damit erzeugte AppManifest-Datei

XAML

```
<Deployment xmlns="http://schemas.microsoft.com/client/2007/deployment"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" EntryPointAssembly="OOB"
EntryPointType="OOB.App" RuntimeVersion="3.0.40818.0">
  <Deployment.OutOfBrowserSettings>
    <OutOfBrowserSettings ShortName="OOB-Anwendung" EnableGPUAcceleration="True"
                          ShowInstallMenuItem="True">
      <OutOfBrowserSettings.Blurb>Eine OOB-Anwendung auf Ihrem Desktop; zu Hause, im Büro
```

BEISPIEL 43.27: Die damit erzeugte AppManifest-Datei

```
                                oder unterwegs.
      </OutOfBrowserSettings.Blurb>
      <OutOfBrowserSettings.WindowSettings>
        <WindowSettings Title="Meine erste OOB-Anwendung" />
      </OutOfBrowserSettings.WindowSettings>
      <OutOfBrowserSettings.Icons>
        <Icon Size="128,128">korganizer.png</Icon>
      </OutOfBrowserSettings.Icons>
    </OutOfBrowserSettings>
  </Deployment.OutOfBrowserSettings>
...
</Deployment>
```

Installation

Nach einem Start Ihrer Anwendung über den Browser steht Ihnen folgendes Kontextmenü zur Verfügung:

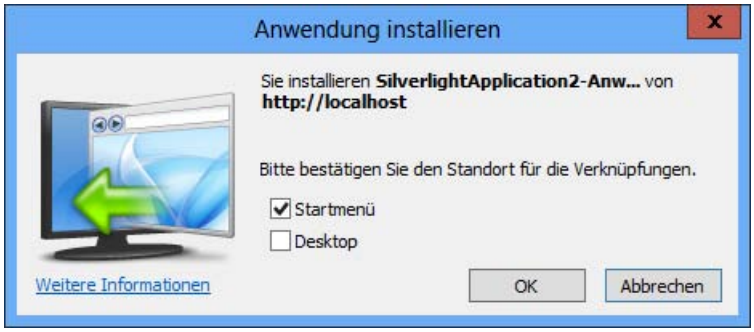
Doch welcher Anwender ist schon so pfiffig und ruft mit der rechten Maustaste einen Menüpunkt auf? Besser Sie bieten aus Ihrer Anwendung heraus eine entsprechende Funktion an. Wer jetzt an umständliche Programmierorgien denkt liegt komplett falsch. Mit einem Methodenaufruf ist der Anwender Ihrer Silverlight-Anwendung "beglückt":

BEISPIEL 43.28: Auslösen per Methode aus dem Silverlight-Plug-in heraus

C#

```
        private void button2_Click(object sender, RoutedEventArgs e)
        {
            Application.Current.Install();
        }
```

Auf einen Fragedialog können Sie verzichten, den zeigt das Silverlight-Plug-in zwangsweise an, um die Sicherheit auf dem Client-PC zu gewährleisten:



Das Icon und die möglichen Optionen sowie die Beschriftungen haben wir vorher in den Projekteigenschaften festgelegt, die Werte werden lediglich aus der *AppManifest*-Datei ausgelesen.

Bestätigt der Anwender den obigen Dialog, findet sich auch eine entsprechende Verknüpfung auf dem Desktop wieder, bei der die heruntergeladenen XAP-Daten mit *sllauncher.exe* gestartet werden:

Das Ganze sieht doch schon fast wie eine richtige Windows-Anwendung aus – oder?

Bemerkungen

Ist Ihre Anwendung abgekoppelt und läuft nicht mehr im Browser, müssen Sie sich natürlich auch um einige Probleme kümmern, die Sie im Zusammenhang mit einer reinen Browseranwendung nicht hätten:

- Arbeite ich noch mit der aktuellen Version und wie komme ich gegebenenfalls an ein Update?
- Ist die Anwendung mit dem Internet verbunden?
- Handelt es sich auch um eine Out-of-Browser-Anwendung?

Die Antwort bzw. die Lösung für obige Fragen finden Sie in den folgenden drei Praxisbeispielen.

43.8.2 Out-of-Browser-Anwendung aktualisieren

Mit der Umwandlung Ihres Silverlight-Programms in eine Out-of-Browser-Anwendung haben Sie zwar dem Endanwender eine Freude gemacht, da die Internet-Verbindung entfallen kann, allerdings müssen Sie sich jetzt selbst darum kümmern, dass die nun lokale Anwendung auch auf dem aktuellen Stand gehalten wird, ein automatisches Update ist in diesem Fall nicht mehr möglich.

HINWEIS: Dieses Beispiel macht nur einen Sinn, wenn es sich um eine Out-of-Browser-Anwendung handelt.

Quellcode

Zwei Varianten bieten sich an: entweder Sie überlassen die Updateprüfung dem Endanwender oder Sie prüfen automatisch bei jedem Programmstart auf neue Versionen (in diesem Fall könnten Sie aber auch gleich die Anwendung im Web belassen).

Wir wollen uns Version 1 näher ansehen:

```
public partial class MainPage : UserControl
{
```

Im Konstruktor des Hauptformulars weisen wir einen Ereignishandler zu, da der eigentliche Updatevorgang, wie so viele Funktionen in Silverlight, asynchron abläuft:

```
    public MainPage()
    {
        InitializeComponent();
        Application.Current.CheckAndDownloadUpdateCompleted +=
                                        myCheckAndDownloadUpdateCompleted;
    }
```

Angestoßen wird die Aktualisierung durch den einfachen Aufruf der Methode *CheckAndDownloadUpdateAsync*:

```
    private void button3_Click(object sender, RoutedEventArgs e)
    {
        Application.Current.CheckAndDownloadUpdateAsync();
    }
```

Nach der mehr oder weniger erfolgreichen Abarbeitung der Methode kommt im finalen Ereignis *CheckAndDownloadUpdateCompleted* auf uns die Aufgabe zu, den per Parameter übermittelten Status dem Endanwender begreiflich zu machen:

```
    void myCheckAndDownloadUpdateCompleted(object sender,
                                        CheckAndDownloadUpdateCompletedEventArgs e)
    {
```

Ein neues Update ist verfügbar:

```
        if (e.UpdateAvailable)
        {
            MessageBox.Show("Update erfolgreich, bitte neu starten ...");
```

```
        }
```

Ein neues Silverlight-Update ist nötig, bevor die neue Version geladen werden kann:

```
        else if (e.Error != null && e.Error is PlatformNotSupportedException)
        {
            MessageBox.Show("Ein Update ist verfügbar, aber es ist eine " +
                            " neuere Version von Silverlight nötig.");
        }
```

Auch das soll es geben:

```
        else
        {
            MessageBox.Show("Kein Update verfügbar!");
        }
    }
```

HINWEIS: Um den Neustart der Anwendung muss sich der Endanwender selbst kümmern, das können wir ihm nicht abnehmen.

43.8.3 Testen auf aktive Internetverbindung

Möchten Sie prüfen, ob Ihre Silverlight-Anwendung momentan mit dem Internet verbunden ist und ob Sie gegebenenfalls Daten nachladen dürfen, können Sie die *NetworkInterface.GetIsNetwork-Available*-Methode aufrufen.

HINWEIS: Dieses Beispiel macht nur einen Sinn, wenn es sich um eine Out-of-Browser-Anwendung handelt.

Quellcode

Zunächst den erforderlichen Namespace einbinden:

```
using System.Net.NetworkInformation;
...
    private void button5_Click(object sender, RoutedEventArgs e)
    {
```

Abfrage starten:

```
        if (NetworkInterface.GetIsNetworkAvailable())
            MessageBox.Show("Netzwerkverbindung vorhanden");
        else
            MessageBox.Show("Keine Netzwerkverbindung vorhanden");
    }
```

HINWEIS: Die obige Methode kann natürlich keine Auskunft über die Erreichbarkeit einzelner Server geben.

43.8.4 Auf Out-of-Browser-Anwendung testen

Haben Sie dem Benutzer die Möglichkeit gegeben, aus der Anwendung eine Out-of-Browser-Anwendung zu machen (siehe Seite 2167) ist es sinnvoll, auch den aktuellen Status der Anwendung zu bestimmen, um beispielsweise Updates etc. nachzuladen.

Quellcode

Die *Application.Current.IsRunningOutOfBrowser*-Eigenschaft liefert uns die gewünschte Information:

```
private void button4_Click(object sender, RoutedEventArgs e)
{
    if (Application.Current.IsRunningOutOfBrowser)
        MessageBox.Show("Out-of-Browser");
    else
        MessageBox.Show("Im Browser");
}
```

43.8.5 Den Browser bestimmen

Auch wenn die Anzahl der von Silverlight unterstützten Zielplattformen und Browser noch überschaubar ist, schadet es sicher nicht wenn es eine Möglichkeit gibt, nähere Auskünfte über die aktuelle Umgebung zu sammeln.

Hier hilft Ihnen die *BrowserInformation*-Eigenschaft weiter, die vom statischen *HtmlPage*-Objekt (Namespace *System.Windows.Browser*) bereitgestellt wird. Folgende Informationen lassen sich abrufen:

Eigenschaften	Beschreibung
BrowserVersion	Version des Browsers.
CookiesEnabled	Dürfen Cookies erzeugt werden?
Name	Der Name des aktuellen Browsers.
Platform	Das umgebende Betriebssystem.
UserAgent	Der User-Agent Header, der vom Browser gesendet wird.

Quellcode

Binden Sie zunächst den folgenden Namespace ein:

```
using System.Windows.Browser;
...
```

Mit dem Laden des Formulars zeigen wir die Informationen in einer *ListBox* an:

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
```

Um die Schreibweise zu vereinfachen, erstellen wir ein neues Objekt:

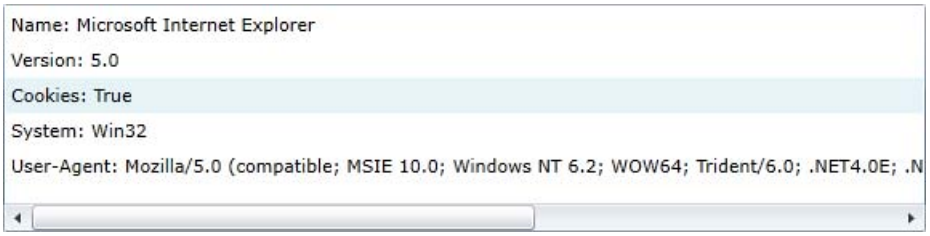
```
            BrowserInformation bi = HtmlPage.BrowserInformation;
```

Abfragen der Werte:

```
            ListBox1.Items.Add("Name: " + bi.Name);
            ListBox1.Items.Add("Version: " + bi.BrowserVersion);
            ListBox1.Items.Add("Cookies: " + bi.CookiesEnabled);
            ListBox1.Items.Add("System: " + bi.Platform);
            ListBox1.Items.Add("User-Agent: " + bi.UserAgent);
        }
    }
}
```

Test

Nach dem Start sollten Sie die gewünschten Informationen in der *ListBox* vorfinden:



HINWEIS: Die meisten Informationen lassen sich dem User-Agent-Header entnehmen.

43.8.6 Parameter an das Plug-in übergeben

Da Ihre Silverlight-Anwendung meist innerhalb eines Browser-Plug-in läuft und dieses wiederum Teil eines größeren Biotops ist, kommt schnell der Wunsch auf, spezielle Parameter an die Silverlight-Anwendung zu übergeben. Auch dies ist kein Problem, wie das folgende Beispiel zeigt.

Oberfläche

Ein Standardseite mit einer *ListBox*.

Quellcode (MainPage.xaml.cs)

Leider stehen uns die gewünschten Parameter nur im *Startup*-Ereignis der *App*-Klasse zur Verfügung, wir müssen also einen Weg finden, wie die Daten an das Formular übergeben werden können.

Dazu fügen wir dem Formular ein *Dictionary* hinzu, das wir im o.g. Ereignis füllen werden:

```
    public partial class MainPage : UserControl
    {
```

```
        public Dictionary<string, string> param;

        public MainPage()
        {
            InitializeComponent();
            param = new Dictionary<string,string>();
        }
```

Hier folgen Auswertung und Anzeige der Parameter:

```
        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
            foreach (KeyValuePair<string, string> p in this.param)
            {
                Listbox1.Items.Add(p.Key + " : " + p.Value);
            }
        }
```

Quellcode (App.xaml.cs)

Im *Startup*-Ereignis übergeben wir die Parameter an unser Formular:

```
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            this.RootVisual = new MainPage();
```

Die Formularinstanz abrufen:

```
            MainPage p = (MainPage) this.RootVisual;
```

Die Ereignisparameter an das Formular übergeben:

```
            p.param = (Dictionary<string,string>) e.InitParams;
        }
```

Quellcode der umgebenden HTML-Seite

Die betreffende Zeile für die Parameterübergabe ist fett hervorgehoben:

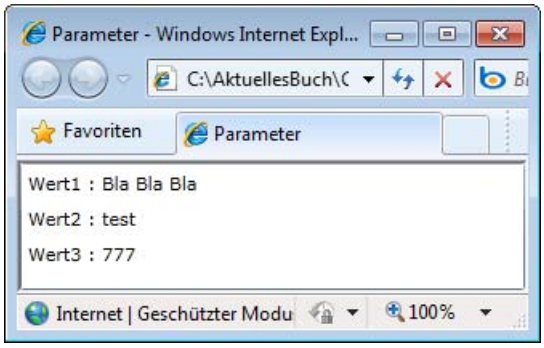
```
...
    <div id="silverlightControlHost">
        <object data="data:application/x-silverlight-2," type="application/x-silverlight-2"
              width="100%" height="100%">
     <param name="source" value="Parameter.xap"/>
     <param name="onError" value="onSilverlightError" />
     <param name="background" value="red" />
     <param name="initparams" value="Wert1=Bla Bla Bla,Wert2=test,Wert3=777" />
     <param name="minRuntimeVersion" value="3.0.40818.0" />
     <param name="autoUpgrade" value="true" />
     <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=3.0.40818.0" style="text-
             decoration:none">
       <img src="http://go.microsoft.com/fwlink/?LinkId=108181" alt="Get Microsoft Silverlight"
             style="border-style:none"/>
```

```
    </a>
    </object><iframe id="_sl_historyFrame"
    style="visibility:hidden;height:0px;width:0px;border:0px"></iframe></div>
```

HINWEIS: Trennen Sie die Wert/Value-Paare mit einem Komma, Wert und Value werden mit einem Gleichheitszeichen zugewiesen!

Test

Nach dem Start sollte für obige Angaben folgende Ausgabe erfolgen:



43.8.7 Auf den QueryString zugreifen

Haben Sie bereits mit Webanwendungen gearbeitet, ist Ihnen sicher auch die Möglichkeit bekannt, Informationen per QueryString zwischen einzelnen Webseiten zu übertragen. Dazu wird an das Ende der normalen Webadresse ein Fragezeichen gefolgt von Key/Value-Paaren angehängt. Die Key/Value-Paare werden durch ein "&"-Zeichen voneinander getrennt.

Oberfläche

Ein Standardseite mit einer *ListBox*.

Quellcode

Namespace einbinden:

```
using System.Windows.Browser;
...
        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
```

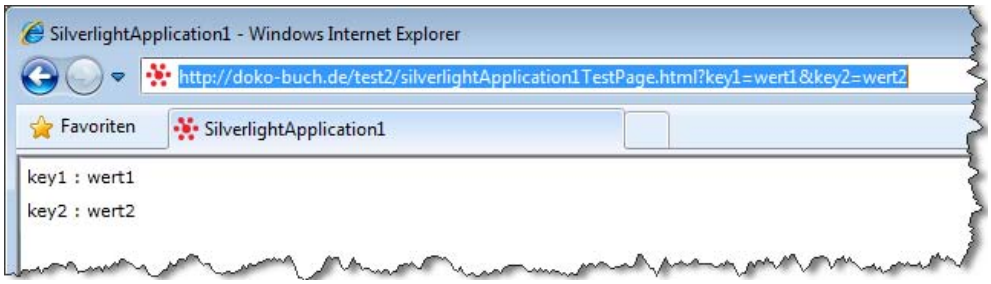
HtmlPage.Document.QueryString ist ein *Dictionary* mit allen Key/Value-Paaren, die als Query-String übergeben wurden:

```
            foreach (KeyValuePair<string, string> p in HtmlPage.Document.QueryString)
            {
```

```
                Listbox1.Items.Add(p.Key + " : " + p.Value);
            }
        }
    }
```

Test

Rufen Sie die Seite mit einigen Parametern auf (siehe zum Beispiel folgende Abbildung) und lassen Sie sich die Werte anzeigen:



43.8.8 Timer in Silverlight nutzen

Ein *Timer*-Control wie in Windows Forms werden Sie im Werkzeugkasten vergeblich suchen, Sie müssen dennoch nicht darauf verzichten, wie folgendes Beispiel zeigt.

Oberfläche

Eine einfache Startseite mit *Loaded*-Ereignis und einem mittig platzierten Label:

```
<UserControl x:Class="Timer_Bsp.MainPage"
...
   Loaded="UserControl_Loaded" >

    <Grid x:Name="LayoutRoot" Background="White">
        <dataInput:Label HorizontalAlignment="Center" Margin="30" Name="label1"
                FontSize="22" />
    </Grid>
</UserControl>
```

Quellcode

Binden Sie zunächst den folgenden Namespace ein, um die Schreibweise etwas zu verkürzen:

```
using System.Windows.Threading;
```

Ist die Seite geladen, erzeugen und starten wir den *Timer*:

```
        private void UserControl_Loaded(object sender, RoutedEventArgs e)
        {
            DispatcherTimer timer1 = new DispatcherTimer();
```

Für das Intervall können Sie verschiedene *TimeSpan*-Konstruktoren nutzen, der folgende bietet die Möglichkeit, Werte in Millisekunden zu übergeben, wie Sie es sicher auch gewohnt sind:

```
timer1.Interval = new TimeSpan(0, 0, 0, 0, 1000);
```

Die Ereignismethode zuweisen:

```
timer1.Tick += new EventHandler(OnTimerTick);
```

Timer starten:

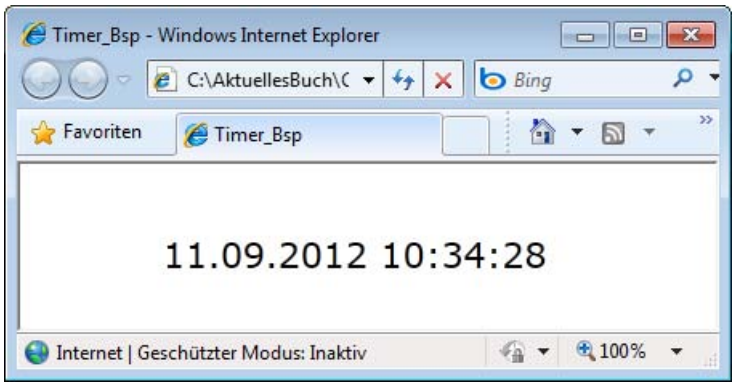
```
timer1.Start();
}
```

Und hier ist das eigentliche Ereignis, das zyklisch aufgerufen wird:

```
public void OnTimerTick(object o, EventArgs sender)
{
    label1.Content = DateTime.Now.ToString();
}
```

Test

Nach dem Start sollte bereits die Uhrzeit angezeigt werden:



43.8.9 Dateien lokal speichern

Neben den im Abschnitt 43.6 beschriebenen Möglichkeiten, Daten in einem isolierten Speicher abzulegen, besteht noch eine weitere Option, die aber das explizite Eingreifen des Nutzers erfordert, d.h., Sie zeigen einen *Datei speichern*-Dialog an und wenn der Nutzer diesen bestätigt, können Sie eine Datei im gewählten Verzeichnis erzeugen.

Oberfläche

Lediglich eine Schaltfläche für das Auslösen der Aktion.

Quellcode

```
...
using System.IO;
```

Fügen Sie der Schaltfläche folgenden Ereignishandler hinzu:

```
        private void button2_Click(object sender, RoutedEventArgs e)
        {
```

Dialog instanziieren:

```
            SaveFileDialog dlg = new SaveFileDialog();
```

Dialog anzeigen und Rückgabewert auswerten:

```
            if (dlg.ShowDialog() == true)
            {
```

Der Dialog gibt uns auch gleich einen *System.IO.Stream* zurück, wenn wir die *OpenFile*-Methode aufrufen:

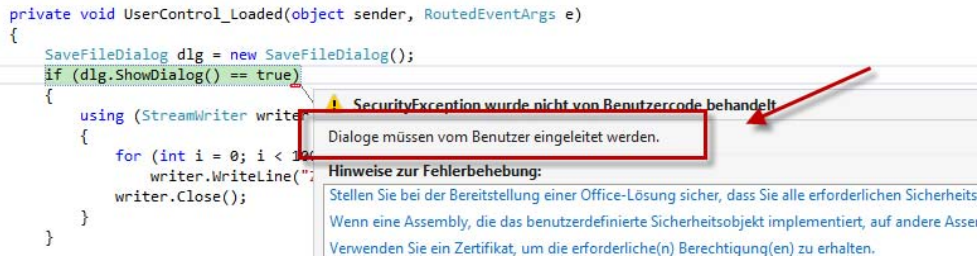
```
                using (StreamWriter writer = new StreamWriter(dlg.OpenFile()))
                {
```

Die Arbeit mit dem guten alten *StreamWriter* dürfte bekannt sein:

```
                    for (int i = 0; i < 100; i++)
                        writer.WriteLine("Zeile " + i.ToString());
                    writer.Close();
                }
            }
        }
```

Bemerkung

Die obige Dialoganzeige muss Ergebnis einer Nutzeraktion (z.B. Button-Klick) sein. Rufen Sie die Routine zum Beispiel aus dem *Loaded*-Ereignis der Seite auf, wird ein Fehler ausgelöst:



HINWEIS: Alternativ können Sie auch mit einer Fulltrust-Anwendung auf alle Nutzerverzeichnisse zugreifen (siehe Abschnitt 43.7, Seite 2164)

43.8.10 Drag & Drop realisieren

Was liegt in vielen Fällen näher, als Daten per Drag & Drop, z.B. aus dem Explorer in eine Anwendung einzufügen? Silverlight unterstützt auch diese Variante des Datenaustausches, wie im folgenden Beispiel gezeigt.

Oberfläche

Fügen Sie lediglich eine *ListBox* in *MainPage.xaml* ein und legen Sie die Eigenschaft *AllowDrop* auf *True* fest.

```
<UserControl x:Class="DragDrop.MainPage"
...
    d:DesignHeight="300" d:DesignWidth="400" >
    <Grid x:Name="LayoutRoot" Background="White">
        <ListBox Name="ListBox1" Drop="ListBox1_Drop_1" AllowDrop="True"/>
    </Grid>
</UserControl>
```

Mit dem *Drop*-Ereignis reagieren wir auf die Nutzeraktion zur Laufzeit.

Quelltext

```
...
using System.IO;
...
```

Die Reaktion auf das "Abwerfen" von Dateien:

```
        private void ListBox1_Drop_1(object sender, DragEventArgs e)
        {
```

Erstmal auf korrekte Daten prüfen:

```
            if (e.Data.GetDataPresent(DataFormats.FileDrop))
            {
```

Die Liste der Dateien abrufen:

```
                FileInfo[] _files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];
```

Die Liste durchlaufen:

```
                foreach (FileInfo _file in _files)
                {
```

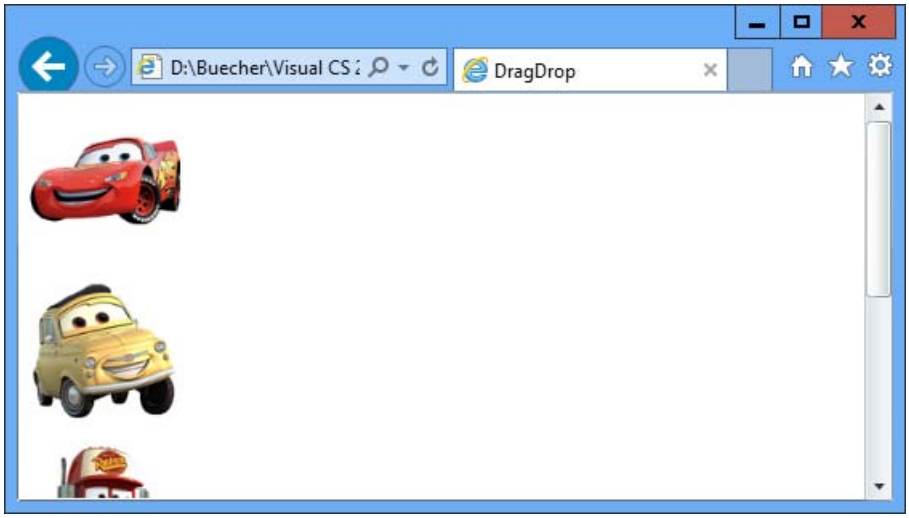
Handelt es sich um eines der folgenden Grafikformate, laden wir die Datei als Bild, andernfalls zeigen wir nur den Dateinamen an:

```
                    if ((".JPG.JPEG.PNG.BMP").Contains(_file.Extension.ToUpper()))
                    {
                        using (FileStream fileStream = _file.OpenRead())
                        {
                            BitmapImage bi = new BitmapImage();
```

```
                        bi.SetSource(fileStream);
                        ListBox1.Items.Add(new Image() { Source = bi, Width = 100 });
                    }
                }
                else
                    ListBox1.Items.Add(_file.Name);
            }
        }
    }
```

Test

Staren Sie die Anwendung und ziehen Sie einige Dateien per Drag & Drop in die *ListBox*. Falls sich Grafikdateien darunter befinden, sollten diese angezeigt werden:



43.8.11 Auf die Zwischenablage zugreifen

Auch wenn die Funktionalität der Zwischenablage unter Silverlight stark eingeschränkt ist – Sie können nur reinen Text kopieren/einfügen – so sollten Sie trotz allem auch diese Funktion in Ihre Anwendung integrieren, wenn es sinnvoll ist.

Oberfläche

Fügen Sie lediglich eine *ListBox* und zwei Schaltflächen in *MainPage.xaml* ein:

```
<UserControl x:Class="Zwischenablage.MainPage"
...    d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <Button Content="Copy" Click="Button_Click_1" />
        <Button Content="Paste" Click="Button_Click_2" />
        <ListBox Name="ListBox1" />
```

```
    </StackPanel>
</UserControl>
```

Quelltext

Mit dem Klick auf die Schaltfläche versuchen wir Text zu kopieren:

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    try
    {
        Clipboard.SetText("Ein kleiner Text");
    }
    catch (Exception)
    {  MessageBox.Show("Kein Zugriff auf die Zwischenablage"); }
}
```

HINWEIS: Wie Sie sehen, haben wir selbst für diese triviale Aufgabe eine Fehlerbehandlung implementiert, der Grund sind die eingeschränkten Rechte unserer Anwendung.

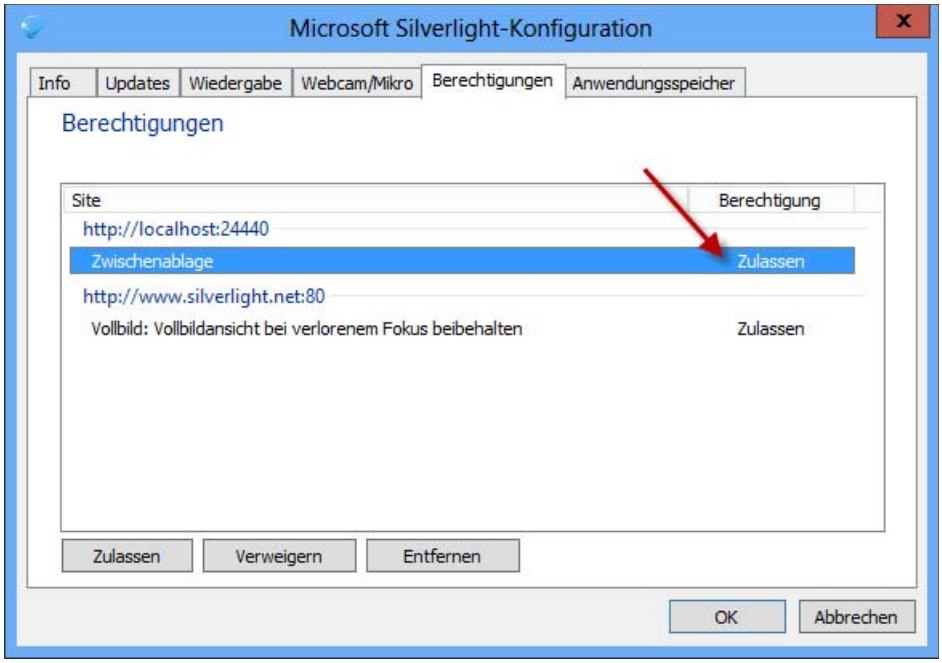
Das Einfügen von Text aus der Zwischenablage:

```
private void Button_Click_2(object sender, RoutedEventArgs e)
{
    if (Clipboard.ContainsText())
    {
        string inhalt = Clipboard.GetText();
        ListBox1.Items.Add(inhalt);
    }
}
```

Test

Starten Sie die Anwendung und klicken Sie auf die Schaltflächen, dürfte folgende Meldung angezeigt werden:

Aus Sicherheitsgründen ist der Zugriff auf die Zwischenablage nur nach vorheriger Nutzerabfrage zulässig. Wird die Option *Antwort speichern* aktiviert, so wird der Anwendung der dauernde Zugriff erlaubt, was Sie auch über den Silverlight-Konfigurationsdialog nachprüfen können.



Hier können Sie das Recht jederzeit auch wieder entziehen.

HINWEIS: Handelt es sich um eine Fulltrust-Anwendung (siehe dazu Abschnitt 43.7, Seite 2164), wird durch das System **keine** Meldung anzeigt!

43.8.12 Weitere Fenster öffnen

Sie sind es leid, bei der Umsetzung von Projekten immer wieder nur mit einem aktiven Fenster arbeiten zu müssen? Wenn ja, finden Sie seit Silverlight 5 eine brauchbare Lösung für dieses Problem in Form der instanziierbaren *Windows*-Klasse

HINWEIS: Voraussetzung für dieses Beispiel ist eine Fulltrust-Anwendung (siehe dazu Abschnitt 43.7, Seite 2164), die als Out-of-Browser-Anwendung läuft.

Oberfläche (MainPage)

Lediglich eine Schaltfläche zum Anzeigen der zusätzlichen Fenster und eine Schaltfläche für die Anzeige der aktuellen Fensterliste.

Quellcode (MainPage.xaml.cs)

```
...
    public partial class MainPage : UserControl
    {
```

Wir blenden ein neues Fenster ein:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
```

Die unbedingten Voraussetzungen testen wir hier ab:

```
            if (Application.Current.IsRunningOutOfBrowser &&
                    Application.Current.HasElevatedPermissions)
            {
```

Die *Window*-Instanz erzeugen (per XAML-Editor können Sie derartige Fenster nicht erzeugen):

```
                Window win = new Window();
```

Window konfigurieren:

```
                win.Width = 450;
                win.Height = 320;
                win.Title = "Ein frei schwebendes Fenster";
                win.WindowState = WindowState.Normal;
                win.TopMost = true;
```

Wir füllen das *Window* mit einem *UserControl*:

```
                SilverlightControl1 c1 = new SilverlightControl1();
```

HINWEIS: Achtung: Die folgenden Eigenschaft *ParentWindow* haben wir dem *UserControl* selbst hinzugefügt, diese benötigen wir unbedingt:

```
                c1.ParentWindow = win;
                win.Content = c1;
```

Window anzeigen:

```
                win.Show();
            }
            else
```

Alternativ eine Fehlermeldung anzeigen:

```
                MessageBox.Show(
                 "Funktioniert nur als Out-of-Browser-Anwendung mit höherer Vertrauensstellung");
        }
    }
```

Die Liste der geöffneten Fenster anzeigen:

```
private void Button_Click_2(object sender, RoutedEventArgs e)
       {
           ListBox1.Items.Clear();
           foreach (var w in Application.Current.Windows)
           {
               ListBox1.Items.Add((w as Window).Title);
           }

       }
```

Oberfläche (SilverlightControl1)

Ein beliebiger Inhalt und eine Schaltfläche zum Schließen des Fensters.

Quellcode (SilverlightControl1.xaml.cs)

```
    public partial class SilverlightControl1 : UserControl
    {
```

Unsere selbstdefinierte Eigenschaft *ParentWindow*:

```
        private Window _parent;
        public Window ParentWindow
        {
            get { return _parent; }
            set { _parent = value; }
        }
```

Nur mit Hilfe dieser Eigenschaft haben wir Zugriff auf das umgebende Fenster des *UserControls* und können so auch das Fenster schließen:

```
        private void Button_Click_1(object sender, RoutedEventArgs e)
        {
            _parent.Close();
        }
...
```

Test

Vergessen Sie nicht über die Projektoptionen den Zieltyp *Out-of-Browser-Anwendung* festzulegen und für diese eine höhere Vertrauensstellung zu fordern. Starten Sie dann die Anwendung und erzeugen Sie zusätzliche Fenster bzw. zeigen Sie die Fensterliste an.

Bemerkungen

Sie haben es sicher schon bemerkt, der Zugriff aus dem UserControl auf das übergeordnete Fenster ist nur mit einem Kunstgriff, d h. einer eigenen Eigenschaft *ParentWindow*, möglich. Das ist sicher nicht der Weisheit letzter Schluss. Auch die Multi-Monitor-Unterstützung lässt zu wünschen übrig. Gleiches gilt für das Erstellen modaler Windows.

Index

.cctor 1850
.NET-Koordinatensystem 1624
.xap-Package 2120

A

Abbrechen 1731
Abstrakte Klasse 1351
AcceptButton 1518, 1563
AcceptsReturn 1564
Access Control List 2077
AccessDataSource 1995
AccessKey 1922
ACL 2077
AcquireRequestState 2041
Activate 1520, 1521
Activator 1446, 1448
ActiveControl 1518
ActiveForm 1518
ActiveLinkColor 1561
ActiveMdiChild 1518
ActiveViewIndex 1947
Add 1235, 1730
AddAfterSelf 1235
AddBeforeSelf 1235
AddFirst 1235
AddNew 1730
AddResource 1819
ADO.NET Entity DataModel 2151
ADO.NET-Klassen 1265
ADO.NET-Objektmodell 1262
AdRotator 1937
AdvertisementFile 1937
AfterSelect 1586
AJAX 2089
Aktionsabfrage 1288
Aktualisieren 1731
AllowFullOpen 1672
AllowPrintToFile 1698
AllowQuit 1500
AllowSelection 1698
AllowSomePages 1698
AllPages 1693
AllPaintingInWmPaint 1769, 1809, 1868
Ancestors 1232
Anchor 1504, 1531, 1590
Anchoring 1530
Animate 1869
Animated Gif 1772
Animationen 1769
AnimGif 1867
Anonyme Anmeldung 2071
Anonyme Methoden 1456
AnonymousTemplate 2084

Antialiasing 1634
App_Data 1902
App.xaml 2124
App.xaml.cs 2124
Appearance 1568, 1569
Append 2163
AppendChild 1218, 1222, 1223
AppendHeader 2051
AppendToLog 2055
Application 2042
Application Package 2120
Application_Error 2063
ApplicationExit 1503
ApplicationPath 2049
ARGB 1653
args 1474
asax.Global 2042
ascx 1902
ASP.NET
  Datenbindung 1922
  E-Mail 2066
  Ereignis-Handler 1909
  Ereignisprogrammierung 1905
  Fehlerbehandlung 1907, 2059
  Fehlerseiten 2061
  Formulare 1913
  Globale Variablen 1913
  Konfiguration 2080
  Label 1921
  Laufzeitfehler 2059
  NET-Projektdateien 1891
  Oberflächengestaltung 1904
  Projektaufbau 1883
  Startformular 1898
  Steuerelemente 1921
  Tabellen 1910
  TextBox 1908, 1923
  Umleiten 1914
  Voraussetzungen 1882
  Web-Export 1917
ASP.NET registrieren 1919
ASP.NET-Webdienste 2139
ASP.NET/AJAX 2114
ASPNetSqlProviders 2079
ASPX-Datei 1894
aspx.cs 1896
Assemblies signieren 1861
Assembly 1858
  dynamisch laden 1446
  GetExecutingAssembly 1444
  Informationen 1858
  Laden 1443
  LoadFrom 1443
AssemblyInfo.cs 1472, 1496
AssemblyKey-File 1861
AssociatedControlID 1922
Attachments 2067, 2070
Attribute 1201, 1204, 1215
Auflistungszuordnung 1370
Aufzählungstyp 1845
Ausgabequalität 1646
Ausschneiden 1439
Auswahl-Listbox 1832
Auswahlabfrage 1291, 1736
Authenticate 2074
AuthenticateRequest 2041
Authentication 2071
Authorization 2077
AuthorizeRequest 2041
AutoComplete 1564
AutoGenerateInsertButton 2025
AutoScroll 1518, 1590
AutoSize 1560
AutoZoom 1702
AvailableFreeSpace 2160

B

BackColor 1504
BackgroundColor 1477
BackgroundImage 1518
base 1851
Basisklassen-Methoden 1852
BaudRate 1449
Beep 1478
BeginPrint 1681
BeginRequest 2041
Benutzerdefiniertes Steuerelement 1831, 1835
Benutzersteuerelement 1831, 1832, 1902
Bind 1923, 2029
Binding 1727
BindingNavigator 1730, 1731
BindingSource 1726
BitBlt 1804
Bitmap 1752, 1813
BitmapData 1752
Bitmaps 1802
BlinkLabel 1835
Body 2067
BodyFormat 2067
Borderstyle 1504, 1560, 1618
BringToFront 1521
Browsable 1841
BrowseDisplayMode 1978
Browser 2049
Browser-Eigenschaften 2048
BrowserInformation 2173
BrowserVersion 2173
Brush 1657
BufferOutput 2051
BulletList 1926, 1927
Button 1562, 1924
Byte-Array 1317

C

CacheDuration 1994
Caching 1933, 1994
Calendar 1931
Callback 1379, 1385
CanAnimate 1774, 1869
CancelButton 1518, 1563
CancelEdit 1731
CanDuplex 1687
Canvas 2105
Category 1842
CausesValidation 1504, 1513, 1967
Change 1513
ChangeDatabase 1271
ChangeMode 2025, 2028
ChangePasswort 2087
Charset 2051
Chart 1597, 1697, 1717, 2036
CheckBox 1567, 1925
CheckBoxList 1926
Checked 1567, 1569
CheckedChanged 1568
CheckedItems 1571
CheckedListbox 1571
CheckItems 1579
CheckState 1567
Childform 1544
ChildNodes 1220
ChildWindow 2129
Class-Konstruktor 1850
ClassesRoot 1442
ClearContent 2051
ClearHeaders 2051
Click 1510
ClickOnce-Deployment 1418
Client-Script 1924
Clientseitige Validierung 1964
ClientSize 1518

Clipboard 1437, 2182
ContainsText 1438
GetImage 1438
SetText 1438
Clipping 1666, 1669
Clone 1300
Close 1270, 1521
Closed 1520
Closing 1520
CodeBehind 1896
CollectionBase 1242
Color 1656
ColorDialog 1671
ColorMatrix 1777
ColumnName 1303
Columns 1301, 1303
COM-Port 1450
ComboBox 1571
Command 1271, 1284
CommandArgument 1947
CommandBuilder 1279
CommandName 1947, 2028, 2030
CommandText 1273
CommandTimeout 1273
CommandType 1274
CommonAppDataPath 1500
CommonAppDataRegistry 1500
CompanyName 1501
CompareDocumentOrder 1232
CompareFieldValidator 1960
CompareValidator 1957
Component 1831
config.Web 2064, 2074
Connection 1266, 1273
ConnectionString 1268
ConnectionStringBuilder 1271
ConnectionTimeout 1270
ConsoleKeyInfo 1480
ConsoleModifiers 1481
Container 1833
Contains 1515
Content-Seiten 1952
ContentEncoding 2051
Contentpage 1973
ContentPlaceHolder 1951
ContentType 2049, 2051
ContextMenu 1504, 1518
Control 1831, 1835, 1836
ControlBox 1518
Controls 1518, 2045
ControlStyles 1769, 1809, 1868, 1870
ControlToValidate 1958
Cookie 1915, 1990
Cookies 2049, 2051, 2056
CookiesEnabled 2173
Copies 1692
Copy 1299
Create 2163
CreateCommand 1271, 1272
CreateControl 1521
CreateDirectory 2161
CreateElement 1218
CreateGraphics 1515, 1521, 1630, 1673, 1767
CreateInstance 1448
CreateMeasurementGraphics 1687
CreateNavigator 1245
CreateNew 2163
CreateSubKey 1442, 1461
CreateUserWizard 2086
Cross Page Posting 1925
CRUD 1312
CultureInfo 1828
CurrentChanged 1375
CurrentConfig 1442
CurrentCulture 1501
CurrentPage 1693
CurrentUICulture 1828
CurrentUser 1442

Cursor 1504, 1518
CursorLeft 1477
CursorTop 1477
CursorVisible 1477
Custom Controls 2095
CustomColors 1672
customErrors 2064
CustomExpression 2015
CustomValidator 1957, 1963

D

DashCap 1656
DashStyle 1656
Data Services 2148
DataAdapter 1283
Database 1269
DataBind 1993, 1995, 2046
DataBindings 1727
DataBits 1449
DataColumn 1301
DataContext 1326, 2005
DataGrid 1597, 1739
DataKeyNames 2020
DataList 1986, 2029
DataMember 1926
DataNavigateUrlField 2021
DataNavigateUrlFormatString 2021
DataReader 1280
DataReceived 1450
DataServiceContext 2151
DataSource 1269, 1986, 1987
DataSourceId 1926, 2019
DataTable 1300
DataTextFormatString 1926
DataValueField 1926
DataView 1305, 1313
Dateianhang 2070
Dateiverknüpfungen 1463
Datenauswahl 1990
Datenbindung 2028
Datenkonsument 1261
Datenprovider 1261, 1262
Datenquelle 1308, 1327, 1732
DateTimePicker 1573
DblClick 1510
DbProviderFactories 1264
Deactivate 1520
DecimalPlaces 1575
Default-Ereignis 1855
Default-Property 1843
DefaultEvent 1855
DefaultMode 2025
DefaultValue 1842
Delegate 1355, 1379, 1394, 1399, 1852
DELETE 1304, 1991
DeleteCommand 1284
DeleteDirectory 2162
DeleteFile 2162
DeleteMethod 1996
DeleteSubKey 1442
DeleteSubKeyTree 1442, 1463
DeleteValue 1442
Deployment 1419
Depth 1248
Descendants 1233
Description 1841
Design Pattern 1379
DesignMode 1863
Designmodus 1863
DesktopBounds 1518
DesktopLocation 1518
Destruktor 1851
DetailsView 1986, 2024
Diagramme 1358
Dialoge 1523
DialogResult 1518
Dictionary 1475

Direction 1278
DirectionVertical 1646
DisabledLinkColor 1561
Dispose 1276
Disposed 2047
DllImport 1792
Dock 1504, 1518, 1530, 1590
Docking 1530
DockPadding 1518, 1531
DOCTYPE 1204
Document 1680, 1698
Document Object Model 1215
Document Type 1215
Document Type Definition 1200
DocumentName 1691
DocumentSource 1936
DoDragDrop 1515, 1521
DoEvents 1501
DOM 1215
Domain 2056
DomainUpDown 1576
Double Buffering 1767
DoubleBuffer 1769, 1810, 1868
Drag & Drop 2180
Drag & Drop-Datenbindung 1730
DragDrop 1513
DragOver 1513
DrawArc 1642
DrawCurve 1638
DrawEllipse 1640
DrawImage 1771, 1777
DrawItemState 1871
DrawLine 1633
DrawPie 1640
DrawPolygon 1637
DrawRectangle 1635
DrawRectangles 1636
DrawString 1644, 1871
DropDownStyle 1572, 1870
Drucker konfigurieren 1707
Druckvorschau 1702
DTD 1200
DtrEnable 1449
Duplex 1692
Dynamisches Modell 1336
DynData 1442

E

e-Parameter 1510
EditDisplayMode 1978
EditItemTemplate 2026
Eigenschaften 1839
Eigenschaftenfenster 1841
Einfügen 1439
Eingabefokus 1909
Eingabereihenfolge 1529
Element 1201, 1204, 1215, 1260
ElementHost 1601, 1609
Elements 1233, 1746
Emboss-Effekt 1766
EMF 1619
EmptyDataTemplate 2026
EmptyDataText 2024
EnableCaching 1994
EnableClientScript 1958
Enabled 1504
EnableViewState 2045
EnableVisualStyles 1492
Encoder.Quality 1779
EncoderParameters 1779
EndEdit 1731
EndElement 1260
Endlicher Zustandsautomat 1359
EndRequest 2042
Enter 1513
EnterThreadModal 1503
Entity Data Model 2007, 2151

Entity-Framework 2148
EntityDataSource 2007
Entwurfszeit-Datenbindung 1728
Enumeration 1347
Environment 1474
Error 404 2066
ErrorMessage 1958
ErrorPage 2045
ErrorReceived 1450
Eval 2029
Event 1379, 1394
Event Pattern 1379, 1404
EventArgs 1852, 1854
EventHandler 1854
Events 1852
ExecutablePath 1501
Execute 2056
ExecuteNonQuery 1272, 1275
ExecuteReader 1272, 1275, 1280
ExecuteScalar 1272, 1275
Exit 1501
ExitThread 1502
ExpandableObjectConverter 1848
ExpandAll 1258
Expires 2056
ExpiresAbsolute 2051

F

Farbauswahl 1671
Fehlerprüfung 1843
FieldCount 1282
File Authorization 2077
FileDropList 1437
FilePath 2049
FileUpload 1936
Fill 1285
FillEllipse 1640
FillMode 1665
FillPolygon 1637
FillRectangle 1635
FillRectangles 1636
FilterExpression 1994
Filtern 1305
FilterParameters 1994
Find 1304
FindControl 2046
FindForm 1515
Finite State Machine 1359
FirstChild 1220, 1254
FLASH 2115
FlatStyle 1562
FlowLayout 1590
FlowLayoutPanel 1533
Fluchtpunktperspektive 1784
Flush 2051
Focus 1515, 1521
Font 1504, 1662
Font-Combobox 1870
FontDialog 1670
FontMustExist 1671
FontStyle 1663
FooterTemplate 2026
for-each 1253
ForeColor 1504
ForegroundColor 1477
Form 1517, 1523, 2049
Form1.cs 1495
Form1.Designer.cs 1494
Format 1573
Format24bppRgb 1750
Format32bppArgb 1750
FormBorderStyle 1518
Forms Authentication 2072
FormView 1986, 2026
FPoint 1649
FrameDimension 1775
FRectangle 1650

FromArgb 1658
FromImage 1813
FSize 1650
FSM 1359
Fulltrust-Anwendung 2164

G

GAC 1861
GDI 1791, 1812
Geerbtes Benutzersteuerelement 1831
Gerätekontext 1794
Gerätekoordinaten 1628
Geschäftsmodell 1384
Geschäftsobjekt 1997
Geschäftsvorfälle 1332
get 1839
GetChanges 1300
GetCommandLineArgs 1474
GetContainerControl 1515
GetDataObject 1437
GetDataPresent 1438
GetDC 1795, 1804
GetDeviceCaps 1687
GetDirectoryNames 2161
GetElementsByTagName 1224
GetEnumerator 1469, 1824
GetEnvironmentVariables 1475
GetFactoryClasses 1264
GetFields 1445
GetFileNames 2161
GetFrameCount 1775
GetHdc 1813
GetImageEncoders 1779
GetLastError 2063
GetManifestResourceNames 1466, 1817
GetManifestResourceStream 1467, 1468, 1818
GetMembers 1445
GetMethod 1448
GetMethods 1445
GetNextControl 1521
GetObject 1820
GetPixel 1632, 1748
GetPortNames 1450, 1458
GetProperties 1445
GetStream 1820
GetSubKeyNames 1442
GetType 1508, 1515
GetTypes 1444
GetUserStoreForApplication 2159
GetUserStoreForSite 2159
GetValue 1282, 1462
GetValueNames 1442
GetValues 1282
GetWindowDC 1803
Gif-Frame 1774
Global.asax 1897
Globale Koordinaten 1625
Globalisierte Anwendungen 1825
Globalization 1828
Grafik
  Auflösung 1620
  Dunkler 1757
  Filter 1763
  Gamma 1760
  Graustufen 1756
  Heller 1757
  Histogramm 1760
  Invertieren 1755
  Kontrast 1759
  maskieren 1777
  Pixelformat 1621
  Skalieren 1623
Grafiktyp 1621
Graphics 1630, 1673
Graphics.FromImage 1630, 1813
GridView 1986, 2000, 2019
GroupBox 1569, 1591

GroupingText 1932

H

Handle 1504
HasControls 2046
HasKeys 2056
HasMorePages 1681
HatchBrush 1658
Hauptformular 1493
Hauptprogramm 1472
HeaderTemplate 2026
HelpButton 1518
HelpProvider 1599
HelpRequested 1513, 1520
HiddenField 1932
Hide 1515, 1521
Hinzufügen 1730
HorizontalResolution 1620
Hotkey 1922
Hotspot 1929
HScrollBar 1574
HTML 1197
HttpApplication 2039, 2040
HTTPServerUtility 2055
Hyperlink 1561, 1929, 2020

I

ICO 1619
Icon 1518
IContainer 1838, 1849
IDataObjekt 1438
Identifikation 2076
IDisposable 1276
Idle 1503
IgnoreCase 1820
IgnoreWhitespace 1244, 1260
Image 1563, 1618, 1752, 1929
ImageAlign 1594
ImageAnimator 1774, 1869
ImageAttributes 1777
ImageButton 1924
ImageCodecInfo 1779
ImageField 2019
ImageFormat 1779, 1813
ImageIndex 1580, 1581
ImageList 1594
ImageLockMode 1752
ImageMap 1929
Images-Collection 1594
ImageSize 1594
Impersonation 2076
ImportRow 1302
Include-Beziehungen 1332
IncreaseQuotaTo 2161
Indent 1251
IndentChars 1251
Indeterminate 1567
Index 2020
Indexer 1843
inherited 1852
Init 2047
InitLayout 1869
InnerException 2063
InnerText 1225
InputStream 2049
INSERT 1991
InsertCommand 1284
InsertItemTemplate 2026
InsertQuery 1991
InsertVisible 2025
InstalledPrinters 1682, 1709
InstallShield 1426
Interface 1353, 1385
InteropServices 1812
IntPtr 1794, 1813
Invalidate 1515, 1521, 1631, 1769

Invoke 1446, 1448, 1455
IObserver 1385
IsAbstract 1445
IsAfter 1233
IsAuthenticated 2049
IsBefore 1233
IsClass 1444
IsClientConnected 2051
IsClosed 1282
IsCOMObject 1445
IsDefaultPrinter 1683
IsEnum 1445
IsInterface 1445
IsMdiChild 1519
IsMdiContainer 1519, 1539
Isolated Storage 2159
IsolatedStorageFileStream 2163
Isolierter Speicher 2159
IsPostBack 1889, 2045
IsPublic 1445
IsSealed 1445
IsSecureConnection 2049
IsValid 1964, 2045
IsWebWebBrowserContextMenuEnabled 1590
Item 1282
Items 1570
Items.Count 1570
ItemTemplate 2027

J

JPEG-Qualität 1779
JScript 1924, 1932

K

Kalender 1573
KeyDown 1511
KeyPress 1511
KeyPreview 1512
KeyUp 1511
Klasse 1349
Klassen-Designer 1344
  Zuordnung 1357
Klassenbibliothek 1857
Klassendiagramm 1345, 1358
Kommentare 1201
Komponenten 1831
Komponentenklasse 1831, 1838
Konsolenanwendung 1450, 1471
Konstruktor 1837, 1849
Kontextmenü 1463
Kopieren 1439

L

Label 1560
Landscape 1686, 1709
Language 1825
LargeChange 1574
LargeImageList 1579, 1581
Layout 1520
LayoutMdi 1541
Leave 1513
LeaveThreadModal 1503
LicFileLicenseProvider 1863, 1864
LinearGradientBrush 1660
Lines 1564
Linien 1633
LinkBehavior 1561
LinkButton 1924
LinkColor 1561
LinkLabel 1561
Links.Add 1561
LINQ to SQL 2005
LINQ to XML-API 1227
LinqDataSource 2004
Listbox 1569

ListView 1578
ListViewItem 1578
Lizenz 1863
Load 1231, 1520
LoadControl 2046
LoadXml 1216
Localizable 1825
LocalMachine 1442
LocalUserAppDataPath 1501
Location 1504, 1519
LocationChanged 1520
LockBits 1749, 1751
Locked 1504
LoggedInTemplate 2084
Login 2083
Login-Dialog 2073
LoginName 2084
LoginStatus 2084
LoginView 2084
Löschen 1730
Low-Level-Grafik 1748
LUT 1757

M

MachineName 2056
Mail-Server 2067
MailMessage 2067, 2070
Main 1473, 1493
MainMenu 1534
MakeTransparent 1778
Manifestressourcen
  Betrachter 1465
  Erstellen 1815
  Zugriff 1817
Manuelle Bindung 1726, 1727
MapPath 2046, 2055
Mapperklassen 1327
Margins 1685, 1690
Marshal 1751, 1752
Mask 1566
MaskCompleted 1567
Master-Detailbeziehung 1739
Masterpage 1901, 1950, 1972
Matrix-Klasse 1745
Matrix33 1777
MatrixOrder 1647
Mausereignisse 1510
Max 1574
MaxDropDownItems 1572
MaximizeBox 1519
MaximumCopies 1692
MaximumPage 1696
MaximumSize 1519
MaxLength 1563
MDI-Applikation 1538
MDIChildActivated 1520
MdiChildren 1519, 1542
MDIContainer 1539
MdiParent 1519, 1540
MeasureString 1645
Membership 2087
MembershipUser 2087
MemoryStream 1317, 1322
Menu 1519, 1938, 1940
Menü 1534
MenuItem 1537
MenuItem.Visible 1537
Metasprache 1197
Methoden 1848
MethodExpression 2015
Microsoft Event Pattern 1379, 1399
Microsoft Word 1705
Min 1574
MinimizeBox 1519
MinimumPage 1696
MinimumSize 1519
Modal 1519

modale Fenster 1523
Modifiers 1481
MonthCalendar 1573
MouseDown 1510
MouseEnter 1511
MouseLeave 1511
MouseMove 1510
MouseUp 1510
MoveBufferArea 1477
MoveToNext 1246
MoveToPrevious 1246
MSComm.ocx 1449
Multicast-Events 1395
MultiExtended 1571
MultiLine 1563
MultiSimple 1571
MultiView 1947

N

Name 1504, 2056
Namespace 1444
NavigateUrl 1929
Navigations-Frame 2136
NET-Reflection 1443
NewRow 1302
NextPageImageUrl 2023
NextSibling 1220, 1254
Nodes 1233
NodeType 1249
Notification 1382
Notification-Sequenz 1383
NumericUpDown 1575
Nutzer erstellen 2081

O

ObjectDataSource 1995
Objekt 1334
Objekt-Eigenschaften 1846
Observer 1380
Observer Pattern 1380
OffsetX 1746
OffsetY 1746
OLE-Automation 1703, 1719
OleDbConnection 1266
OnClientClick 1924
OnPaint 1673, 1771, 1869
Opacity 1519, 1528
Open 1270, 2163
OpenForms 1501
OpenOrCreate 2163
OpenSubKey 1462
OrderByExpression 2015
Out-of-Browser-Applikation 2167
OutputStream 2051
OwnedForms 1519
Owner 1519, 1544

P

Page 2045, 2134
Page_Error 2062
Page_Load 1913
PagerSettings 2023
PagerTemplate 2027
PageScale 1626
PageSetupDialog 1690, 1699
PageUnit 1629, 1689
Paging 2019, 2022
Paint 1513, 1520, 1630, 1673
Panel 1590, 1932
PaperKind 1685
PaperName 1684
PaperSizes 1684, 1709
Parallelisierung 1765
Parallelprojektion 1783
Parameter 1277, 1990, 1991, 2174

ParameterName 1278
Parameterübergabe 1474
Parent 1538
Parentform 1544
ParentNode 1254
Parity 1449
Parser 1204
PasswordChar 1565
PasswortRecovery 2085
Path 1663, 2057
PathGradientBrush 1661
Pen 1654
PenType 1656
PerformanceData 1442
PI 1204
Picture.Image 1618
PictureBox 1572, 1768
Pinsel 1657
Pixel 1632
PixelFormat 1621
Platform 2173
Point 1649
Pointer 1753
PointToClient 1522
PointToScreen 1522
Polygon 1637
PolyLine 1635
Portalseite 1975
PortName 1449
PostbackUrl 1925
PostBackValue 1931
PostRequestHandlerExecute 2041
PreRender 2047
PreRequestHandlerExecute 2041
PreviousPageImageUrl 2023
PreviousSibling 1254
PrintDialog 1689, 1698
PrintDocument 1679
PrinterName 1689, 1709
PrinterResolution 1686
PrinterResolutions 1709
PrinterSettings 1698
PrintPage 1679, 1681, 1690
PrintPreviewDialog 1680, 1701
PrintRange 1693
PrintToFile 1698
Processing Instructions 1200, 1204, 1215
ProductName 1501
ProductVersion 1501
Program 1472
Program.cs 1471
Progressbar 1576
ProhibitDtd 1260
Prolog 1200
Properties.Resources 1497
Properties.Settings 1499
PropertyExpression 2015
PropertyGrid 1872
Provider 1266, 1269

Q

Query Builder 1989
QueryExtender 2015
QueryPageSettings 1681, 1689
QueryString 1990, 2020, 2049, 2050, 2176
Quota 2160

R

RadioButton 1925
RadioButtonList 1926
RangeExpression 2015
RangeValidator 1957, 1962
Rawformat 1621
ReadContentAsFloat 1250
ReadKey 1477, 1480
ReadOnly 1563

ReadTimeout 1449
ReadXml 1239, 1317, 1322
Rechte einschränken 2082
Rectangle 1650
RectangleToClient 1522
RectangleToScreen 1522
Redirect 2051
RedirectFromLoginPage 2074
Refresh 1515, 1522, 1751, 1995
Regedit.exe 1440
Region 1666, 1667
Registration-Sequenz 1382
Registrierungsdatenbank 1440
Registrierungseditor 1440
Registry 1439, 1442, 1461
RegistryKey 1440, 1442, 1461
RegularExpressionValidator 1957, 1962
ReleaseDC 1803, 1804
ReleaseHdc 1813
ReleaseRequestState 2041
Remove 1236, 1304, 1730
RemoveAll 1236
RemoveAnnotations 1236
RemoveAt 1730
RemoveAttributes 1236
RemoveContent 1236
RemoveCurrent 1730
RemoveFilter 1730
RepeatColumns 2032
RepeatDirection 2032
Repeater 2032
Request 2039, 2048
RequestType 2049
RequiredFieldValidator 1957, 1959
Resize 1513, 1520
ResolveRequestCache 2041
ResolveUrl 2046
ResourceManager 1820
ResourceReader 1469, 1824
Resources 1819
Resources resx 1497
ResourceWriter 1819
Response 2039, 2051, 2074
Ressourcen 1861
Ressourcen auflisten 1817
RGB 1755
RichTextBox 1577
Roles 2087
Rotate 1806
RotateFlip 1622
RotateTransform 1627, 1647
Rotation 1627, 1747, 1787
RowFilter 1305
Rows 1303
RS232 1449
Run 1492, 1502

S

Save 1619, 1813
Scale 1515, 1806
ScaleTransform 1627
Scan0 1749, 1751, 1752
Scherung 1747
Schnittstelle 1353
Schriftauswahl 1670
Screenshot 1812
Scripting 1932
ScrollBars 1563
SearchExpession 2015
Secure 2057
Seitenkoordinaten 1626
Seitennavigation 1938
Seitentransformation 1628
SELECT 1246, 1291, 1564, 1736
Selectable 1870
SelectActiveFrame 1775
SelectAll 1564

SelectCommand 1284
SelectedIndex 1570, 1572, 1871
SelectedIndexChanged 1570, 1593, 2020
SelectedItem 1570, 1871
SelectedText 1564
Selection 1693
SelectionLength 1564
SelectionMode 1570, 1571, 1572, 1931
SelectionStart 1564
SelectMethod 1996
SelectNextControl 1515, 1522
SelectNodes 1224
SelectSingleNode 1221, 1222, 1223, 1224
Send 1514
sender 1508
SendKeys 1514
SendWait 1514
Sequenzdiagramme 1382
SerialPort 1449
Serielle Schnittstelle 1449
Server 2039, 2055
Serverseitige Validierung 1964
ServerVariables 2049
ServerVersion 1269
Session 1913, 1990, 2001, 2040, 2043
Session beenden 2044
Session-Ereignisse 2044
Session-Variablen 2043
set 1839
SetAttributeValue 1235
SetBounds 1515
SetBufferSize 1477
SetClip 1667
SetColorMatrix 1777
SetCompatibleTextRenderingDefault 1492
SetCursorPosition 1477
SetDataObject 1437
SetElementValue 1235
SetFocusOnError 1967
SetPixel 1632, 1748
SetSmoothingMode 1634
SetStyle 1769, 1809, 1868
Setup-Projekt 1426
SetValue 1461
SetViewportExtEx 1796
SetWindowExtEx 1796
SetWindowOrgEx 1798
SetWindowPosition 1477
SetWindowSize 1477
SetWorldTransform 1800
Shear 1747, 1806
Show 1515, 1522, 1523
ShowDialog 1522, 1523, 1670
ShowHelp 1698
ShowInTaskbar 1519
ShowNetwork 1698
Sicherheitseinstellungen 2087
Silverlight 2109
Silverlight-Anwendung 2111, 2117
Silverlight-Fenster 2129
Silverlight-Navigationsanwendung 2111
Silverlight.js 2120
Site 2040
SiteMap 1901, 1941
SiteMap.Web 1938, 2003
SitemapDataSource 2003
siteMapNode 1939
SiteMapPath 1938, 1946
Size 1519, 1650
SizeMode 1572, 1617
Skalieren 1649
Skalierung 1626, 1746, 1786
Skin 1953, 1967
Skip 1234
SkipWhile 1234
SmallChange 1574
SmallImageList 1579
SmoothingModeHighQuality 1634

SmoothingModeHighSpeed 1634
SmtpClient 2067
SolidBrush 1656, 1658
SomePages 1693
Sort 1305
Sorted 1570, 1572
Sortieren 1305
Sorting 1580
Spline 1638
SplitButton 1595
SplitContainer 1533, 1590
SQLDataSource 1988
Stammelement 1202
Standard-Eigenschaft 1843
Standardargument 1853
StartCap 1656
Startobjekt 1493
StartPosition 1519
Startup 2125
StartupPath 1501
State 1270
StateImageList 1579
stateless 1999
Statisches Modell 1335
StopBits 1449
StoredProcedure 1274
StreamReader 2163
StretchBlt 1805
Stride 1751
StringFormat 1646
StringReader 1317
Strong Name 1860
Struktur 1351
Style Sheet 1955, 1969
Subject 2067
Subjekt 1380
SubKeyCount 1442
SubmitChanges 1329
Substitution 1933
Suchen 1306
SupportsColor 1686
System.Console 1476
System.Drawing 1615
System.Drawing.Design 1617
System.Drawing.Drawing2D 1616, 1745
System.Drawing.Imaging 1616
System.Drawing.Printing 1616
System.Drawing.Text 1617
System.IO.IsolatedStorage 2159
System.IO.Ports 1449
System.Reflection 1443
System.Runtime 1812
System.Xml 1216, 1255
System.Xml.Linq 1227
System.Xml.Serialization 1244
System.Xml.XPath 1248
System.Xml.Xsl 1252
SystemBrushes 1651
SystemColors 1652
SystemIcons 1652
SystemPens 1651

T

TabControl 1592
TabIndex 1504
Table 1927
Table-Layout 2031
TableAdapter 1309
TableCell 1927
TableDirect 1274
TableLayout 1590
TableLayoutPanel 1533
TableName 1303
TableRow 1927
TabPages 1592
TabStop 1504
Tabulatorreihenfolge 1529

Tabulatortaste 1529
Tag 1504
Take 1234
TakeWhile 1234
Target 1929
Tastaturabfrage 1480
Terminalprogramm 1456
Text 1504, 1519, 1563, 1572
TextAlign 1563
TextAlignment 1560
Textausgabe 1644
TextBox 1563
TextChanged 1923
Texteigenschaften 1645
TextMode 1923
TextRenderingHint 1635, 1646
TextureBrush 1659
Themes 1953, 1967
this 1522
ThousandsSeparator 1575
ThreadException 1503
ThreadExit 1503
Threading 1828
Thumbnail 1621
Tick 1599
TickCount 1767
Timer 1599, 2177
ToolStrip 1595
ToolTip 1599
TopMost 1519
TotalBytes 2049
Trackbar 1575
Transfer 2056
Transform 1252, 1748, 2013
Transformationsdatei 1252, 1935
Transformationsmatrix 1806
TransformFile 2013
TransformSource 1936
Translate 1806
TranslateTransform 1626
Translation 1626, 1746, 1785
TransparencyKey 1519
Transparenz 1528, 1776
TreeView 1257, 1584, 1585, 1938, 1943
Truncate 2163
Type 1444
TypeConverter 1848
Typisierte DataSets 1306
Typisierte Ressourcen 1819

U

Uhrzeit 1573
UI-Thread 1456
Umgebungsvariablen 1475
UML 1331
UnicodeEncoding 1317
Unload 2047
UnlockBits 1751
Unregistration-Sequenz 1383
unsafe 1753
Unterverzeichnisse erzeugen 2161
UPDATE 1279, 1286, 1991
UpdateCommand 1284
UpdateFrames 1774, 1870
UpdateRequestCache 2041
Updates 1421
URI 1590
Url 2049
URL-Authorization 2077
Use Case-Diagramm 1331
Use Case-Dokumentation 1333
UseAntialias 1701
User 2040
User Controls 2095
UserAgent 2173
UserAppDataPath 1501
UserAppDataRegistry 1501

UserControl 1831, 2131
UserHostAddress 2049
UserHostName 2049
UserLanguages 2049
UserPaint 1769, 1809, 1868
Users 1442
UTF-16 1204
UTF-8 1204

V

Validate 1513, 2046
ValidateUser 2088
ValidationGroup 1967
ValidationSummary 1957, 1966
Validator 1957, 2045
Value 1278, 1574, 2057
ValueCount 1442
Values 2057
VBScript 1924
Vektorgrafik 1780
Vererbung 1358
Veröffentlichen 1423
VerticalResolution 1620
Vertrauensstellung 2165
View 1579, 1947
Visible 1504, 1519
VisitedLinkColor 1561
Vorfahrtyp 1836
Vorschaugrafik 1621
VScrollBar 1574

W

W3C 1215
WCF-Dienste 2139
Web.config 1898, 2077
Web.sitemap 1901
Webbenutzersteuerelemente 2095
WebBrowser 1589
WebPartManager 1975
Webparts 1956, 1975
WebPartZone 1975
Webpublishing-Assistent 1419, 1425
Webseitenlayout 1950
Website administrieren 2079
Weichzeichnen 1766
Werkzeugkasten 1346
Wertebereichsbeschränkung 1843
Where 1234, 2013
Win32 1856
Windows Authentication 2072
Windows-Messages 1855
Windows-Steuerelementebibliothek 1857
WindowState 1519
Wizard 1948
WMF 1619
WndProc 1856
WordWrap 1564
WParam 1856
WPF
  Silverlight-Applikationstypen 2110
  Zielplattformen 2110
WPF-ClickOnce 2114
Wrap 1932
Write 1478, 2051
WriteAttributeString 1250
WriteByte 1752
WriteEndDocument 1251
WriteEndElement 1250
WriteFile 2051
WriteLine 1478
WriteStartDocument 1250
WriteStartElement 1250
WriteXml 1317, 1322
WriteXmlSchema 1239
WWW-Dienst 1918

X

XAttribute 1229
XComment 1229
XDeclaration 1229
XDocument 1228, 1230
XDocumentType 1229
XElement 1228
XElement.Load 1231
XElement.Parse 1231
XML 1197, 1934
XML transformieren 1236
XML-Schema 1210
XmlAttribute 1215, 1244
XMLCDATASection 1216
XMLCharacterData 1216
XMLComment 1216
XmlDataDocument 1239
XmlDataSource 2013
XmlDocument 1239, 1245, 1254
XmlDocumentType 1216
XmlElement 1216, 1244
XmlEntity 1216
XmlEnum 1244
XmlIgnore 1244
XmlImplementation 1216
XMLNamedNodeMap 1215
XmlNode 1215, 1219
XMLNodeList 1215
XmlNodeType 1260
XMLParseError 1215
XmlProcessingInstructions 1216
XmlReader 1248, 1259
XmlReaderSettings 1248, 1260
XmlRoot 1244
XmlSerializer 1242
XmlText 1216
XmlTextWriter 1317
XmlWriter 1250
XmlWriterSettings 1251
XNode 1229
XPath 2013
XPathDocument 1245
XPathNavigator 1245
XPathNodeIterator 1246
XProcessingInstruction 1229
XSD-Schema 1208
xsd.exe 1214
XslCompiledTransform 1252
XSLT 1252, 1934

Z

Zeichenoperationen 1675
Zeilenauswahl 2019
Zentralprojektion 1784
Zustände 1336
Zustandsänderung 1381
Zustandsüberführungsdiagramm 1337
Zustandsüberführungsmatrix 1359
Zwischenablage 2181

VISUAL C# 2012 //

- **Für Einsteiger, Fortgeschrittene und Profis**
- **Mit Informationen und Lösungen, nach denen Sie anderswo vergeblich suchen**
- **EXTRA: Kostenloses E-Book inkl. 1.000 Seiten Bonuskapitel**
- **Alle Beispiele aus dem Buch unter: www.doko-buch.de**

Sprachumfang und Einsatzgebiete von Visual C# decken mit jeder neuen Version ein immer breiteres, kaum mehr überschaubares Spektrum ab. Dieses Übungs- und Nachschlagewerk führt Sie mit sicherer Hand quer durch den Dschungel der .NET-Programmierung auf Basis von Visual C# 2012.

Schwerpunkt dieser Neuauflage ist die Entwicklung von Apps für Windows 8. Neben einer fundierten Einführung erleichtern viele Praxisbeispiele den Einstieg in dieses komplexe Thema.

Die insgesamt 43 Kapitel des Buches gruppieren sich in acht Themenkomplexe:

Teil I: Grundlagen der Programmierung mit Visual C# 2012

Teil II: Technologien der Programmentwicklung

Teil III: WPF-Anwendungen

Teil IV: Windows Store Apps

Teil V: Weitere Technologien

Teil VI: Windows Forms

Teil VII: ASP.NET

Teil VIII: Silverlight

Die Kapitel innerhalb eines Teils bilden eine logische Aufeinanderfolge, können aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Walter **DOBERENZ** und Thomas **GEWINNUS** haben sich als Entwickler, Dozenten und Fachbuchautoren für Microsoft- und Borland-Entwicklerwerkzeuge einen Namen gemacht, ihre Fachbücher sind seit Jahren als Standardwerke etabliert.

AUS DEM INHALT //

- **C#-Sprachfeatures, OOP, LINQ**
- **Dateien & Dateisystem**
- **Asynchrone Programmierung & Task Parallel Library**
- **WPF & XAML**
- **Windows Store Apps & WinRT**
- **XML, ADO.NET, DataSet, MS Event Pattern**
- **Konsolen-Anwendungen**
- **GUI-Programmierung mit Windows Forms**
- **Web-Applikationen mit ASP.NET & Silverlight**
- **Verteilen von Anwendungen**

UNSER BUCHTIPP FÜR SIE //

Doberenz/Gewinnus, Visual C# 2012 – Kochbuch
2013. 850 Seiten. FlexCover. € 39,90.
ISBN 978-3-446-43438-7

HANSER

€ 49,90 [D] | € 51,30 [A]
ISBN 978-3-446-43439-4

9 783446 434394

www.hanser-fachbuch.de/computer