walter DOBERENZ
thomas GEWINNUS

Visual Basic 2015

GRUNDLAGEN
PROFIWISSEN
REZEPTE

// Visual Basic-Grundlagen
// LINQ, OOP, ADO.NET
// App-Entwicklung
// Über 150 Praxisbeispiele

EXTRA: 700 Seiten Bonuskapitel zu WPF und Windows Forms

HANSER

Doberenz/Gewinnus

Visual Basic 2015
Grundlagen, Profiwissen und Rezepte

Bleiben Sie auf dem Laufenden!

Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter

www.hanser-fachbuch.de/newsletter

Hanser Update ist der IT-Blog des Hanser Verlags mit Beiträgen und Praxistipps von unseren Autoren rund um die Themen Online Marketing, Webentwicklung, Programmierung, Softwareentwicklung sowie IT- und Projektmanagement. Lesen Sie mit und abonnieren Sie unsere News unter

www.hanser-fachbuch.de/update

Walter Doberenz
Thomas Gewinnus

Visual Basic 2015

Grundlagen, Profiwissen und Rezepte

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.

© 2015 Carl Hanser Verlag München
http://www.hanser-fachbuch.de

Lektorat: Sieglinde Schärl
Herstellung: Irene Weilhart
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

Print-ISBN: 978-3-446-44380-8
E-Book-ISBN: 978-3-446-44605-2

Inhaltsverzeichnis

Vorwort **45**

Teil I: Grundlagen

1 Einstieg in Visual Studio 2015 ..... **51**

1.1 Die Installation von Visual Studio 2015 51
1.1.1 Überblick über die Produktpalette 51
1.1.2 Anforderungen an Hard- und Software 52
1.2 Unser allererstes VB-Programm 53
1.2.1 Vorbereitungen 53
1.2.2 Programm schreiben 55
1.2.3 Programm kompilieren und testen 55
1.2.4 Einige Erläuterungen zum Quellcode 56
1.2.5 Konsolenanwendungen sind out 57
1.3 Die Windows-Philosophie 57
1.3.1 Mensch-Rechner-Dialog 58
1.3.2 Objekt- und ereignisorientierte Programmierung 58
1.3.3 Windows-Programmierung unter Visual Studio 2015 59
1.4 Die Entwicklungsumgebung Visual Studio 2015 61
1.4.1 Neues Projekt 61
1.4.2 Die wichtigsten Fenster 62
1.5 Microsofts .NET-Technologie 64
1.5.1 Zur Geschichte von .NET 64
1.5.2 .NET-Features und Begriffe 66
1.6 Wichtige Neuigkeiten in Visual Studio 2015 74
1.6.1 Entwicklungsumgebung 74
1.6.2 Neue VB-Sprachfeatures 74
1.6.3 Code-Editor 74
1.6.4 NET Framework 4.6 75

1.7 Praxisbeispiele ... 75
1.7.1 Windows-Anwendung für Einsteiger ... 75
1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger ... 79

2 Einführung in Visual Basic ... 87

2.1 Grundbegriffe ... 87
2.1.1 Anweisungen ... 87
2.1.2 Bezeichner ... 88
2.1.3 Kommentare ... 89
2.1.4 Zeilenumbruch ... 90
2.2 Datentypen, Variablen und Konstanten ... 92
2.2.1 Fundamentale Typen ... 92
2.2.2 Wertetypen versus Verweistypen ... 93
2.2.3 Benennung von Variablen ... 93
2.2.4 Deklaration von Variablen ... 94
2.2.5 Typinferenz ... 97
2.2.6 Konstanten deklarieren ... 97
2.2.7 Gültigkeitsbereiche von Deklarationen ... 98
2.2.8 Lokale Variablen mit Dim ... 98
2.2.9 Lokale Variablen mit Static ... 99
2.2.10 Private globale Variablen ... 99
2.2.11 Public Variablen ... 100
2.3 Wichtige Datentypen im Überblick ... 100
2.3.1 Byte, Short, Integer, Long ... 100
2.3.2 Single, Double und Decimal ... 101
2.3.3 Char und String ... 101
2.3.4 Date ... 102
2.3.5 Boolean ... 103
2.3.6 Object ... 103
2.4 Konvertieren von Datentypen ... 104
2.4.1 Implizite und explizite Konvertierung ... 104
2.4.2 Welcher Datentyp passt zu welchem? ... 105
2.4.3 Konvertierungsfunktionen ... 106
2.4.4 CType-Funktion ... 107
2.4.5 Konvertieren von Strings ... 107
2.4.6 Die Convert-Klasse ... 109
2.4.7 Die Parse-Methode ... 109
2.4.8 Boxing und Unboxing ... 110

2.4.9 TryCast-Operator ... 111
2.4.10 Nullable Types ... 111
2.5 Operatoren ... 112
2.5.1 Arithmetische Operatoren ... 112
2.5.2 Zuweisungsoperatoren ... 113
2.5.3 Logische Operatoren ... 114
2.5.4 Vergleichsoperatoren ... 115
2.5.5 Rangfolge der Operatoren ... 115
2.6 Kontrollstrukturen ... 116
2.6.1 Verzweigungsbefehle ... 116
2.6.2 Schleifenanweisungen ... 119
2.7 Benutzerdefinierte Datentypen ... 120
2.7.1 Enumerationen ... 120
2.7.2 Strukturen ... 121
2.8 Nutzerdefinierte Funktionen/Prozeduren ... 124
2.8.1 Deklaration und Syntax ... 124
2.8.2 Parameterübergabe allgemein ... 126
2.8.3 Übergabe mit ByVal und ByRef ... 127
2.8.4 Optionale Parameter ... 128
2.8.5 Überladene Funktionen/Prozeduren ... 128
2.9 Praxisbeispiele ... 129
2.9.1 Vom PAP zum Konsolen-Programm ... 129
2.9.2 Vom Konsolen- zum Windows-Programm ... 131
2.9.3 Schleifenanweisungen kennen lernen ... 133
2.9.4 Methoden überladen ... 136
2.9.5 Eine Iterationsschleife verstehen ... 138
2.9.6 Anwendungen von C# nach Visual Basic portieren ... 141

3 OOP-Konzepte ... 149

3.1 Strukturierter versus objektorientierter Entwurf ... 149
3.1.1 Was bedeutet strukturierte Programmierung? ... 149
3.1.2 Was heißt objektorientierte Programmierung? ... 150
3.2 Grundbegriffe der OOP ... 151
3.2.1 Objekt, Klasse, Instanz ... 151
3.2.2 Kapselung und Wiederverwendbarkeit ... 152
3.2.3 Vererbung und Polymorphie ... 152
3.2.4 Sichtbarkeit von Klassen und ihren Mitgliedern ... 153
3.2.5 Allgemeiner Aufbau einer Klasse ... 154

3.3 Ein Objekt erzeugen ... 155
3.3.1 Referenzieren und Instanziieren ... 156
3.3.2 Klassische Initialisierung ... 157
3.3.3 Objekt-Initialisierer ... 157
3.3.4 Arbeiten mit dem Objekt ... 157
3.3.5 Zerstören des Objekts ... 158
3.4 OOP-Einführungsbeispiel ... 158
3.4.1 Vorbereitungen ... 158
3.4.2 Klasse definieren ... 159
3.4.3 Objekt erzeugen und initialisieren ... 160
3.4.4 Objekt verwenden ... 160
3.4.5 Unterstützung durch die IntelliSense ... 160
3.4.6 Objekt testen ... 161
3.4.7 Warum unsere Klasse noch nicht optimal ist ... 162
3.5 Eigenschaften ... 162
3.5.1 Eigenschaften kapseln ... 162
3.5.2 Eigenschaften mit Zugriffsmethoden kapseln ... 165
3.5.3 Lese-/Schreibschutz für Eigenschaften ... 166
3.5.4 Statische Eigenschaften ... 167
3.5.5 Selbst implementierende Eigenschaften ... 168
3.6 Methoden ... 169
3.6.1 Öffentliche und private Methoden ... 169
3.6.2 Überladene Methoden ... 170
3.6.3 Statische Methoden ... 171
3.7 Ereignisse ... 172
3.7.1 Ereignise deklarieren ... 172
3.7.2 Ereignis auslösen ... 173
3.7.3 Ereignis auswerten ... 173
3.7.4 Benutzerdefinierte Ereignisse (Custom Events) ... 175
3.8 Arbeiten mit Konstruktor und Destruktor ... 178
3.8.1 Der Konstruktor erzeugt das Objekt ... 178
3.8.2 Bequemer geht's mit einem Objekt-Initialisierer ... 180
3.8.3 Destruktor und Garbage Collector räumen auf ... 181
3.8.4 Mit Using den Lebenszyklus des Objekts kapseln ... 184
3.9 Vererbung und Polymorphie ... 184
3.9.1 Vererbungsbeziehungen im Klassendiagramm ... 184
3.9.2 Überschreiben von Methoden (Method-Overriding) ... 186
3.9.3 Klassen implementieren ... 186

3.9.4 Objekte implementieren ... 191
3.9.5 Ausblenden von Mitgliedern durch Vererbung ... 192
3.9.6 Allgemeine Hinweise und Regeln zur Vererbung ... 194
3.9.7 Polymorphe Methoden ... 195
3.10 Besondere Klassen und Features ... 197
3.10.1 Abstrakte Klassen ... 197
3.10.2 Abstrakte Methoden ... 198
3.10.3 Versiegelte Klassen ... 198
3.10.4 Partielle Klassen ... 199
3.10.5 Die Basisklasse System.Object ... 201
3.10.6 Property-Accessors ... 202
3.10.7 Nullbedingter Operator ... 202
3.11 Schnittstellen (Interfaces) ... 203
3.11.1 Definition einer Schnittstelle ... 203
3.11.2 Implementieren einer Schnittstelle ... 204
3.11.3 Abfragen, ob eine Schnittstelle vorhanden ist ... 205
3.11.4 Mehrere Schnittstellen implementieren ... 205
3.11.5 Schnittstellenprogrammierung ist ein weites Feld ... 205
3.12 Praxisbeispiele ... 206
3.12.1 Eigenschaften sinnvoll kapseln ... 206
3.12.2 Eine statische Klasse anwenden ... 209
3.12.3 Vom fetten zum dünnen Client ... 211
3.12.4 Schnittstellenvererbung verstehen ... 220
3.12.5 Aggregation und Vererbung gegenüberstellen ... 224
3.12.6 Eine Klasse zur Matrizenrechnung entwickeln ... 230
3.12.7 Rechner für komplexe Zahlen ... 236
3.12.8 Formel-Rechner mit dem CodeDOM ... 244
3.12.9 Einen Funktionsverlauf grafisch darstellen ... 249
3.12.10 Sortieren mit IComparable/IComparer ... 253
3.12.11 Objektbäume in generischen Listen abspeichern ... 258
3.12.12 OOP beim Kartenspiel erlernen ... 264

4 Arrays, Strings, Funktionen ... 269

4.1 Datenfelder (Arrays) ... 269
4.1.1 Ein Array deklarieren ... 269
4.1.2 Zugriff auf Array-Elemente ... 270
4.1.3 Oberen Index ermitteln ... 270
4.1.4 Explizite Arraygrenzen ... 270

4.1.5 Arrays erzeugen und initialisieren ... 270
4.1.6 Zugriff mittels Schleife ... 271
4.1.7 Mehrdimensionale Arrays ... 272
4.1.8 Dynamische Arrays ... 273
4.1.9 Zuweisen von Arrays ... 274
4.1.10 Arrays aus Strukturvariablen ... 275
4.1.11 Löschen von Arrays ... 276
4.1.12 Eigenschaften und Methoden von Arrays ... 276
4.1.13 Übergabe von Arrays ... 279
4.2 Zeichenkettenverarbeitung ... 280
4.2.1 Strings zuweisen ... 280
4.2.2 Eigenschaften und Methoden eines Strings ... 280
4.2.3 Kopieren eines Strings in ein Char-Array ... 283
4.2.4 Wichtige (statische) Methoden der String-Klasse ... 283
4.2.5 Die StringBuilder-Klasse ... 285
4.3 Reguläre Ausdrücke ... 288
4.3.1 Wozu braucht man reguläre Ausdrücke? ... 288
4.3.2 Eine kleine Einführung ... 289
4.3.3 Wichtige Methoden der Klasse Regex ... 289
4.3.4 Kompilierte reguläre Ausdrücke ... 291
4.3.5 RegexOptions-Enumeration ... 292
4.3.6 Metazeichen (Escape-Zeichen) ... 293
4.3.7 Zeichenmengen (Character Sets) ... 294
4.3.8 Quantifizierer ... 295
4.3.9 Zero-Width Assertions ... 296
4.3.10 Gruppen ... 300
4.3.11 Text ersetzen ... 300
4.3.12 Text splitten ... 301
4.4 Datums- und Zeitberechnungen ... 302
4.4.1 Grundlegendes ... 302
4.4.2 Wichtige Eigenschaften von DateTime-Variablen ... 303
4.4.3 Wichtige Methoden von DateTime-Variablen ... 304
4.4.4 Wichtige Mitglieder der DateTime-Struktur ... 305
4.4.5 Konvertieren von Datumstrings in DateTime-Werte ... 306
4.4.6 Die TimeSpan-Struktur ... 306
4.5 Vordefinierten Funktionen ... 308
4.5.1 Mathematik ... 308
4.5.2 Datums- und Zeitfunktionen ... 310

4.6 Zahlen formatieren ... 312
4.6.1 Die ToString-Methode ... 313
4.6.2 Die Format-Methode ... 314
4.6.3 Stringinterpolation ... 316
4.7 Praxisbeispiele ... 316
4.7.1 Zeichenketten verarbeiten ... 316
4.7.2 Zeichenketten mittels StringBuilder addieren ... 319
4.7.3 Reguläre Ausdrücke testen ... 323
4.7.4 Fehler bei mathematischen Operationen behandeln ... 325
4.7.5 Methodenaufrufe mit Array-Parametern ... 328
4.7.6 String in Array kopieren und umgekehrt ... 331
4.7.7 Ein Byte-Array in einen String konvertieren ... 333
4.7.8 Strukturvariablen in Arrays einsetzen ... 335

5 Weitere Sprachfeatures ... 339

5.1 Namespaces (Namensräume) ... 339
5.1.1 Ein kleiner Überblick ... 339
5.1.2 Die Imports-Anweisung ... 341
5.1.3 Namespace-Alias ... 341
5.1.4 Namespaces in Projekteigenschaften ... 342
5.1.5 Namespace Alias Qualifizierer ... 343
5.1.6 Eigene Namespaces einrichten ... 343
5.2 Überladen von Operatoren ... 344
5.2.1 Syntaxregeln ... 344
5.2.2 Praktische Anwendung ... 345
5.2.3 Konvertierungsoperatoren überladen ... 346
5.3 Auflistungen (Collections) ... 347
5.3.1 Beziehungen zwischen den Schnittstellen ... 347
5.3.2 IEnumerable ... 348
5.3.3 ICollection ... 349
5.3.4 IList ... 349
5.3.5 Iteratoren ... 349
5.3.6 Die ArrayList-Collection ... 350
5.3.7 Die Hashtable ... 351
5.4 Generische Datentypen ... 352
5.4.1 Wie es früher einmal war ... 352
5.4.2 Typsicherheit durch Generics ... 354
5.4.3 List-Collection ersetzt ArrayList ... 355

5.4.4 Über die Vorzüge generischer Collections ... 356
5.4.5 Typbeschränkungen durch Constraints ... 357
5.4.6 Collection-Initialisierer ... 358
5.4.7 Generische Methoden ... 358
5.5 Delegates ... 359
5.5.1 Delegates sind Methodenzeiger ... 359
5.5.2 Delegate-Typ definieren ... 360
5.5.3 Delegate-Objekt erzeugen ... 361
5.5.4 Delegates vereinfacht instanziieren ... 361
5.5.5 Relaxed Delegates ... 362
5.5.6 Anonyme Methoden ... 362
5.5.7 Lambda-Ausdrücke ... 363
5.5.8 Lambda-Ausdrücke in der Task Parallel Library ... 364
5.6 Dynamische Programmierung ... 366
5.6.1 Wozu dynamische Programmierung? ... 366
5.6.2 Das Prinzip der dynamischen Programmierung ... 366
5.6.3 Kovarianz und Kontravarianz ... 370
5.7 Weitere Datentypen ... 371
5.7.1 BigInteger ... 371
5.7.2 Complex ... 373
5.7.3 Tuple(Of T) ... 374
5.7.4 SortedSet(Of T) ... 374
5.8 Praxisbeispiele ... 376
5.8.1 ArrayList versus generische List ... 376
5.8.2 Delegates und Lambda Expressions ... 379
5.8.3 Mit dynamischem Objekt eine Datei durchsuchen ... 382

6 Einführung in LINQ ... 387

6.1 LINQ-Grundlagen ... 387
6.1.1 Die LINQ-Architektur ... 387
6.1.2 LINQ-Implementierungen ... 388
6.1.3 Anonyme Typen ... 388
6.1.4 Erweiterungsmethoden ... 390
6.2 Abfragen mit LINQ to Objects ... 391
6.2.1 Grundlegendes zur LINQ-Syntax ... 391
6.2.2 Zwei alternative Schreibweisen von LINQ-Abfragen ... 392
6.2.3 Übersicht der wichtigsten Abfrage-Operatoren ... 394

6.3 LINQ-Abfragen im Detail ... 395
6.3.1 Die Projektionsoperatoren Select und SelectMany ... 396
6.3.2 Der Restriktionsoperator Where ... 398
6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy ... 398
6.3.4 Der Gruppierungsoperator GroupBy ... 400
6.3.5 Verknüpfen mit Join ... 401
6.3.6 Aggregat-Operatoren ... 402
6.3.7 Verzögertes Ausführen von LINQ-Abfragen ... 404
6.3.8 Konvertierungsmethoden ... 405
6.3.9 Abfragen mit PLINQ ... 405
6.4 Praxisbeispiele ... 408
6.4.1 Die Syntax von LINQ-Abfragen verstehen ... 408
6.4.2 Aggregat-Abfragen mit LINQ ... 411
6.4.3 LINQ im Schnelldurchgang erlernen ... 413
6.4.4 Strings mit LINQ abfragen und filtern ... 416
6.4.5 Duplikate aus einer Liste oder einem Array entfernen ... 417
6.4.6 Arrays mit LINQ initialisieren ... 420
6.4.7 Arrays per LINQ mit Zufallszahlen füllen ... 422
6.4.8 Einen String mit Wiederholmuster erzeugen ... 423
6.4.9 Mit LINQ Zahlen und Strings sortieren ... 425
6.4.10 Mit LINQ Collections von Objekten sortieren ... 426
6.4.11 Ergebnisse von LINQ-Abfragen in ein Array kopieren ... 428

Teil II: Technologien

7 Zugriff auf das Dateisystem ... 431
7.1 Grundlagen ... 431
7.1.1 Klassen für Verzeichnis- und Dateioperationen ... 432
7.1.2 Statische versus Instanzen-Klasse ... 432
7.2 Übersichten ... 433
7.2.1 Methoden der Directory-Klasse ... 433
7.2.2 Methoden eines DirectoryInfo-Objekts ... 434
7.2.3 Eigenschaften eines DirectoryInfo-Objekts ... 434
7.2.4 Methoden der File-Klasse ... 434
7.2.5 Methoden eines FileInfo-Objekts ... 435
7.2.6 Eigenschaften eines FileInfo-Objekts ... 436

7.3 Operationen auf Verzeichnisebene ... 436
7.3.1 Existenz eines Verzeichnisses/einer Datei feststellen ... 436
7.3.2 Verzeichnisse erzeugen und löschen ... 437
7.3.3 Verzeichnisse verschieben und umbenennen ... 438
7.3.4 Aktuelles Verzeichnis bestimmen ... 438
7.3.5 Unterverzeichnisse ermitteln ... 438
7.3.6 Alle Laufwerke ermitteln ... 439
7.3.7 Dateien kopieren und verschieben ... 440
7.3.8 Dateien umbenennen ... 441
7.3.9 Dateiattribute feststellen ... 441
7.3.10 Verzeichnis einer Datei ermitteln ... 443
7.3.11 Alle im Verzeichnis enthaltene Dateien ermitteln ... 443
7.3.12 Dateien und Unterverzeichnisse ermitteln ... 443
7.4 Zugriffsberechtigungen ... 444
7.4.1 ACL und ACE ... 444
7.4.2 SetAccessControl-Methode ... 445
7.4.3 Zugriffsrechte anzeigen ... 445
7.5 Weitere wichtige Klassen ... 446
7.5.1 Die Path-Klasse ... 446
7.5.2 Die Klasse FileSystemWatcher ... 447
7.6 Datei- und Verzeichnisdialoge ... 449
7.6.1 OpenFileDialog und SaveFileDialog ... 449
7.6.2 FolderBrowserDialog ... 451
7.7 Praxisbeispiele ... 452
7.7.1 Infos über Verzeichnisse und Dateien gewinnen ... 452
7.7.2 Die Verzeichnisstruktur in eine TreeView einlesen ... 455
7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen ... 457

8 Dateien lesen und schreiben ... 463

8.1 Grundprinzip der Datenpersistenz ... 463
8.1.1 Dateien und Streams ... 463
8.1.2 Die wichtigsten Klassen ... 464
8.1.3 Erzeugen eines Streams ... 465
8.2 Dateiparameter ... 465
8.2.1 FileAccess ... 465
8.2.2 FileMode ... 465
8.2.3 FileShare ... 466

8.3 Textdateien ... 466
8.3.1 Eine Textdatei beschreiben bzw. neu anlegen ... 466
8.3.2 Eine Textdatei lesen ... 468
8.4 Binärdateien ... 470
8.4.1 Lese-/Schreibzugriff ... 470
8.4.2 Die Methoden ReadAllBytes und WriteAllBytes ... 470
8.4.3 BinaryReader/BinaryWriter erzeugen ... 471
8.5 Sequenzielle Dateien ... 471
8.5.1 Lesen und Schreiben von strukturierten Daten ... 471
8.5.2 Serialisieren von Objekten ... 472
8.6 Dateien verschlüsseln und komprimieren ... 473
8.6.1 Das Methodenpärchen Encrypt-/Decrypt ... 474
8.6.2 Verschlüsseln unter Windows Vista/7/8/10 ... 474
8.6.3 Verschlüsseln mit der CryptoStream-Klasse ... 475
8.6.4 Dateien komprimieren ... 476
8.7 Memory Mapped Files ... 477
8.7.1 Grundprinzip ... 477
8.7.2 Erzeugen eines MMF ... 478
8.7.3 Erstellen eines Map View ... 478
8.8 Praxisbeispiele ... 479
8.8.1 Auf eine Textdatei zugreifen ... 479
8.8.2 Einen Objektbaum speichern ... 483
8.8.3 Ein Memory Mapped File (MMF) verwenden ... 490
8.8.4 Hex-Dezimal-Bytes-Konverter ... 492
8.8.5 Eine Datei verschlüsseln ... 496
8.8.6 Eine Datei komprimieren ... 499
8.8.7 Echte ZIP-Dateien erstellen ... 501
8.8.8 PDFs erstellen/exportieren ... 502
8.8.9 Eine CSV-Datei erstellen ... 506
8.8.10 Eine CSV-Datei mit LINQ lesen und auswerten ... 509
8.8.11 Einen korrekten Dateinamen erzeugen ... 511

9 Asynchrone Programmierung ... 513

9.1 Übersicht ... 513
9.1.1 Multitasking versus Multithreading ... 514
9.1.2 Deadlocks ... 515
9.1.3 Racing ... 515

9.2 Programmieren mit Threads ... 517
9.2.1 Einführungsbeispiel ... 517
9.2.2 Wichtige Thread-Methoden ... 518
9.2.3 Wichtige Thread-Eigenschaften ... 520
9.2.4 Einsatz der ThreadPool-Klasse ... 521
9.3 Sperrmechanismen ... 523
9.3.1 Threading ohne SyncLock ... 523
9.3.2 Threading mit SyncLock ... 524
9.3.3 Die Monitor-Klasse ... 527
9.3.4 Mutex ... 530
9.3.5 Methoden für die parallele Ausführung sperren ... 531
9.3.6 Semaphore ... 532
9.4 Interaktion mit der Programmoberfläche ... 533
9.4.1 Die Werkzeuge ... 534
9.4.2 Einzelne Steuerelemente mit Invoke aktualisieren ... 534
9.4.3 Mehrere Steuerelemente aktualisieren ... 535
9.4.4 Ist ein Invoke-Aufruf nötig? ... 536
9.4.5 Und was ist mit WPF? ... 536
9.5 Timer-Threads ... 538
9.6 Die BackgroundWorker-Komponente ... 539
9.7 Asynchrone Programmier-Entwurfsmuster ... 542
9.7.1 Kurzübersicht ... 542
9.7.2 Polling ... 543
9.7.3 Callback verwenden ... 544
9.7.4 Callback mit Parameterübergabe verwenden ... 545
9.7.5 Callback mit Zugriff auf die Programm-Oberfläche ... 546
9.8 Asynchroner Aufruf beliebiger Methoden ... 547
9.8.1 Die Beispielklasse ... 547
9.8.2 Asynchroner Aufruf ohne Callback ... 549
9.8.3 Asynchroner Aufruf mit Callback und Anzeigefunktion ... 549
9.8.4 Aufruf mit Rückgabewerten (per Eigenschaft) ... 550
9.8.5 Aufruf mit Rückgabewerten (per EndInvoke) ... 551
9.9 Es geht auch einfacher – Async und Await ... 552
9.9.1 Der Weg von synchron zu asynchron ... 552
9.9.2 Achtung: Fehlerquellen! ... 554
9.9.3 Eigene asynchrone Methoden entwickeln ... 556

9.10 Praxisbeispiele ... 558
9.10.1 Spieltrieb & Multithreading erleben ... 558
9.10.2 Prozess- und Thread-Informationen gewinnen ... 570
9.10.3 Ein externes Programm starten ... 575

10 Die Task Parallel Library ... 579

10.1 Überblick ... 579
10.1.1 Parallel-Programmierung ... 579
10.1.2 Möglichkeiten der TPL ... 582
10.1.3 Der CLR-Threadpool ... 582
10.2 Parallele Verarbeitung mit Parallel.Invoke ... 583
10.2.1 Aufrufvarianten ... 584
10.2.2 Einschränkungen ... 585
10.3 Verwendung von Parallel.For ... 585
10.3.1 Abbrechen der Verarbeitung ... 587
10.3.2 Auswerten des Verarbeitungsstatus ... 588
10.3.3 Und was ist mit anderen Iterator-Schrittweiten? ... 588
10.4 Collections mit Parallel.ForEach verarbeiten ... 589
10.5 Die Task-Klasse ... 590
10.5.1 Einen Task erzeugen ... 590
10.5.2 Task starten ... 591
10.5.3 Datenübergabe an den Task ... 592
10.5.4 Wie warte ich auf das Taskende? ... 593
10.5.5 Tasks mit Rückgabewerten ... 595
10.5.6 Die Verarbeitung abbrechen ... 598
10.5.7 Fehlerbehandlung ... 602
10.5.8 Weitere Eigenschaften ... 602
10.6 Zugriff auf das Userinterface ... 604
10.6.1 Task-Ende und Zugriff auf die Oberfläche ... 604
10.6.2 Zugriff auf das UI aus dem Task heraus ... 605
10.7 Weitere Datenstrukturen im Überblick ... 607
10.7.1 Threadsichere Collections ... 607
10.7.2 Primitive für die Threadsynchronisation ... 608
10.8 Parallel LINQ (PLINQ) ... 608
10.9 Praxisbeispiel: Spieltrieb – Version 2 ... 609
10.9.1 Aufgabenstellung ... 609
10.9.2 Global-Klasse ... 609
10.9.3 Controller ... 610

10.9.4 LKWs ... 612
10.9.5 Schiff-Klasse ... 613
10.9.6 Oberfläche ... 615

11 Fehlersuche und Behandlung ... 617

11.1 Der Debugger ... 617
11.1.1 Allgemeine Beschreibung ... 617
11.1.2 Die wichtigsten Fenster ... 618
11.1.3 Debugging-Optionen ... 621
11.1.4 Praktisches Debugging am Beispiel ... 623
11.2 Arbeiten mit Debug und Trace ... 627
11.2.1 Wichtige Methoden von Debug und Trace ... 627
11.2.2 Besonderheiten der Trace-Klasse ... 630
11.2.3 TraceListener-Objekte ... 631
11.3 Caller Information ... 634
11.3.1 Attribute ... 634
11.3.2 Anwendung ... 634
11.4 Fehlerbehandlung ... 635
11.4.1 Anweisungen zur Fehlerbehandlung ... 635
11.4.2 Try-Catch ... 635
11.4.3 Try-Finally ... 640
11.4.4 Das Standardverhalten bei Ausnahmen festlegen ... 642
11.4.5 Die Exception-Klasse ... 643
11.4.6 Fehler/Ausnahmen auslösen ... 643
11.4.7 Eigene Fehlerklassen ... 644
11.4.8 Exceptionhandling zur Entwurfszeit ... 646
11.4.9 Code Contracts ... 646

12 XML in Theorie und Praxis ... 649

12.1 XML – etwas Theorie ... 649
12.1.1 Übersicht ... 649
12.1.2 Der XML-Grundaufbau ... 652
12.1.3 Wohlgeformte Dokumente ... 653
12.1.4 Processing Instructions (PI) ... 656
12.1.5 Elemente und Attribute ... 656
12.1.6 Verwendbare Zeichensätze ... 658

12.2 XSD-Schemas ... 660
12.2.1 XSD-Schemas und ADO.NET ... 660
12.2.2 XML-Schemas in Visual Studio analysieren ... 662
12.2.3 XML-Datei mit XSD-Schema erzeugen ... 665
12.2.4 XSD-Schema aus einer XML-Datei erzeugen ... 666
12.3 XML-Integration in Visual Basic ... 667
12.3.1 XML-Literale ... 667
12.3.2 Einfaches Navigieren durch späte Bindung ... 670
12.3.3 Die LINQ to XML-API ... 672
12.3.4 Neue XML-Dokumente erzeugen ... 673
12.3.5 Laden und Sichern von XML-Dokumenten ... 675
12.3.6 Navigieren in XML-Daten ... 677
12.3.7 Auswählen und Filtern ... 679
12.3.8 Manipulieren der XML-Daten ... 679
12.3.9 XML-Dokumente transformieren ... 681
12.4 Verwendung des DOM unter .NET ... 684
12.4.1 Übersicht ... 684
12.4.2 DOM-Integration in Visual Basic ... 685
12.4.3 Laden von Dokumenten ... 685
12.4.4 Erzeugen von XML-Dokumenten ... 686
12.4.5 Auslesen von XML-Dateien ... 688
12.4.6 Direktzugriff auf einzelne Elemente ... 689
12.4.7 Einfügen von Informationen ... 690
12.4.8 Suchen in den Baumzweigen ... 692
12.5 Weitere Möglichkeiten der XML-Verarbeitung ... 696
12.5.1 Die relationale Sicht mit XmlDataDocument ... 696
12.5.2 XML-Daten aus Objektstrukturen erzeugen ... 699
12.5.3 Schnelles Suchen in XML-Daten mit XPathNavigator ... 702
12.5.4 Schnelles Auslesen von XML-Daten mit XmlReader ... 705
12.5.5 Erzeugen von XML-Daten mit XmlWriter ... 707
12.5.6 XML transformieren mit XSLT ... 709
12.6 Praxisbeispiele ... 711
12.6.1 Mit dem DOM in XML-Dokumenten navigieren ... 711
12.6.2 XML-Daten in eine TreeView einlesen ... 714
12.6.3 DataSets in XML-Strings konvertieren ... 718
12.6.4 In Dokumenten mit dem XPathNavigator navigieren ... 722

13 Einführung in ADO.NET ... 727

13.1 Eine kleine Übersicht ... 727
13.1.1 Die ADO.NET-Klassenhierarchie ... 727
13.1.2 Die Klassen der Datenprovider ... 728
13.1.3 Das Zusammenspiel der ADO.NET-Klassen ... 731
13.2 Das Connection-Objekt ... 732
13.2.1 Allgemeiner Aufbau ... 732
13.2.2 OleDbConnection ... 732
13.2.3 Schließen einer Verbindung ... 734
13.2.4 Eigenschaften des Connection-Objekts ... 734
13.2.5 Methoden des Connection-Objekts ... 736
13.2.6 Der ConnectionStringBuilder ... 737
13.3 Das Command-Objekt ... 738
13.3.1 Erzeugen und Anwenden eines Command-Objekts ... 738
13.3.2 Erzeugen mittels CreateCommand-Methode ... 739
13.3.3 Eigenschaften des Command-Objekts ... 739
13.3.4 Methoden des Command-Objekts ... 741
13.3.5 Freigabe von Connection- und Command-Objekten ... 742
13.4 Parameter-Objekte ... 744
13.4.1 Erzeugen und Anwenden eines Parameter-Objekts ... 744
13.4.2 Eigenschaften des Parameter-Objekts ... 744
13.5 Das CommandBuilder-Objekt ... 745
13.5.1 Erzeugen ... 745
13.5.2 Anwenden ... 746
13.6 Das DataReader-Objekt ... 746
13.6.1 DataReader erzeugen ... 747
13.6.2 Daten lesen ... 747
13.6.3 Eigenschaften des DataReaders ... 748
13.6.4 Methoden des DataReaders ... 748
13.7 Das DataAdapter-Objekt ... 749
13.7.1 DataAdapter erzeugen ... 749
13.7.2 Command-Eigenschaften ... 750
13.7.3 Fill-Methode ... 751
13.7.4 Update-Methode ... 752
13.8 Praxisbeispiele ... 753
13.8.1 Wichtige ADO.NET-Objekte im Einsatz ... 753
13.8.2 Eine Aktionsabfrage ausführen ... 755
13.8.3 Eine Auswahlabfrage aufrufen ... 757

13.8.4 Die Datenbank aktualisieren ... 759
13.8.5 Den ConnectionString speichern ... 762

14 Das DataSet ... 765

14.1 Grundlegende Features des DataSets ... 765
14.1.1 Die Objekthierarchie ... 766
14.1.2 Die wichtigsten Klassen ... 766
14.1.3 Erzeugen eines DataSets ... 767
14.2 Das DataTable-Objekt ... 769
14.2.1 DataTable erzeugen ... 769
14.2.2 Spalten hinzufügen ... 769
14.2.3 Zeilen zur DataTable hinzufügen ... 770
14.2.4 Auf den Inhalt einer DataTable zugreifen ... 771
14.3 Die DataView ... 773
14.3.1 Erzeugen eines DataView ... 773
14.3.2 Sortieren und Filtern von Datensätzen ... 773
14.3.3 Suchen von Datensätzen ... 774
14.4 Typisierte DataSets ... 774
14.4.1 Ein typisiertes DataSet erzeugen ... 775
14.4.2 Das Konzept der Datenquellen ... 776
14.4.3 Typisierte DataSets und TableAdapter ... 777
14.5 Die Qual der Wahl ... 778
14.5.1 DataReader – der schnelle Lesezugriff ... 779
14.5.2 DataSet – die Datenbank im Hauptspeicher ... 779
14.5.3 Objektrelationales Mapping – die Zukunft? ... 780
14.6 Praxisbeispiele ... 781
14.6.1 Im DataView sortieren und filtern ... 781
14.6.2 Suche nach Datensätzen ... 783
14.6.3 Ein DataSet in einen XML-String serialisieren ... 784
14.6.4 Untypisierte in typisierte DataSets konvertieren ... 789
14.6.5 Eine LINQ to SQL-Abfrage ausführen ... 794

15 Verteilen von Anwendungen ... 799

15.1 ClickOnce-Deployment ... 800
15.1.1 Übersicht/Einschränkungen ... 800
15.1.2 Die Vorgehensweise ... 801
15.1.3 Ort der Veröffentlichung ... 801
15.1.4 Anwendungsdateien ... 802

15.1.5 Erforderliche Komponenten ... 802
15.1.6 Aktualisierungen ... 803
15.1.7 Veröffentlichungsoptionen ... 804
15.1.8 Veröffentlichen ... 805
15.1.9 Verzeichnisstruktur ... 805
15.1.10 Der Webpublishing-Assistent ... 807
15.1.11 Neue Versionen erstellen ... 808
15.2 InstallShield ... 808
15.2.1 Installation ... 808
15.2.2 Aktivieren ... 809
15.2.3 Ein neues Setup-Projekt ... 809
15.2.4 Finaler Test ... 817
15.3 Hilfedateien programmieren ... 817
15.3.1 Der HTML Help Workshop ... 818
15.3.2 Bedienung am Beispiel ... 819
15.3.3 Hilfedateien in die VB-Anwendung einbinden ... 821
15.3.4 Eine alternative Hilfe-IDE verwenden ... 825

16 Weitere Techniken ... 827

16.1 Zugriff auf die Zwischenablage ... 827
16.1.1 Das Clipboard-Objekt ... 827
16.1.2 Zwischenablage-Funktionen für Textboxen ... 829
16.2 Arbeiten mit der Registry ... 829
16.2.1 Allgemeines ... 830
16.2.2 Registry-Unterstützung in .NET ... 831
16.3 .NET-Reflection ... 833
16.3.1 Übersicht ... 833
16.3.2 Assembly laden ... 833
16.3.3 Mittels GetType und Type Informationen sammeln ... 834
16.3.4 Dynamisches Laden von Assemblies ... 836
16.4 Praxisbeispiele ... 838
16.4.1 Zugriff auf die Registry ... 838
16.4.2 Dateiverknüpfungen erzeugen ... 840
16.4.3 Die Zwischenablage überwachen und anzeigen ... 842
16.4.4 Die WIA-Library kennenlernen ... 845
16.4.5 Auf eine Webcam zugreifen ... 857
16.4.6 Auf den Scanner zugreifen ... 859
16.4.7 OpenOffice.org Writer per OLE steuern ... 863

16.4.8 Nutzer und Gruppen des Systems ermitteln ... 871
16.4.9 Testen, ob Nutzer in einer Gruppe enthalten ist ... 872
16.4.10 Testen, ob der Nutzer ein Administrator ist ... 874
16.4.11 Die IP-Adressen des Computers bestimmen ... 875
16.4.12 Die IP-Adresse über den Hostnamen bestimmen ... 876
16.4.13 Diverse Systeminformationen ermitteln ... 877
16.4.14 Sound per MCI aufnehmen ... 886
16.4.15 Mikrofonpegel anzeigen ... 889
16.4.16 Pegeldiagramm aufzeichnen ... 891
16.4.17 Sound-und Video-Dateien per MCI abspielen ... 895

17 Konsolenanwendungen ... 903

17.1 Grundaufbau/Konzepte ... 903
17.1.1 Unser Hauptprogramm – Module1.vb ... 904
17.1.2 Rückgabe eines Fehlerstatus ... 905
17.1.3 Parameterübergabe ... 906
17.1.4 Zugriff auf die Umgebungsvariablen ... 907
17.2 Die Kommandozentrale: System.Console ... 908
17.2.1 Eigenschaften ... 908
17.2.2 Methoden/Ereignisse ... 909
17.2.3 Textausgaben ... 910
17.2.4 Farbangaben ... 911
17.2.5 Tastaturabfragen ... 912
17.2.6 Arbeiten mit Streamdaten ... 913
17.3 Praxisbeispiel: Farbige Konsolenanwendung ... 914

Teil III: Windows Apps

18 Erste Schritte ... 919

18.1 Grundkonzepte und Begriffe ... 919
18.1.1 Windows Runtime (WinRT) ... 919
18.1.2 Windows Store Apps ... 920
18.1.3 Fast and Fluid ... 921
18.1.4 Process Sandboxing und Contracts ... 922
18.1.5 .NET WinRT-Profil ... 924
18.1.6 Language Projection ... 924
18.1.7 Vollbildmodus – war da was? ... 926

18.1.8 Windows Store ... 926
18.1.9 Zielplattformen ... 927
18.2 Entwurfsumgebung ... 928
18.2.1 Betriebssystem ... 928
18.2.2 Windows-Simulator ... 928
18.2.3 Remote-Debugging ... 931
18.3 Ein (kleines) Einstiegsbeispiel ... 932
18.3.1 Aufgabenstellung ... 932
18.3.2 Quellcode ... 932
18.3.3 Oberflächenentwurf ... 935
18.3.4 Installation und Test ... 937
18.3.5 Touchscreen ... 939
18.3.6 Fazit ... 939
18.4 Weitere Details zu WinRT ... 941
18.4.1 Wo ist WinRT einzuordnen? ... 942
18.4.2 Die WinRT-API ... 943
18.4.3 Wichtige WinRT-Namespaces ... 945
18.4.4 Der Unterbau ... 946
18.5 Praxisbeispiel ... 948
18.5.1 WinRT in Desktop-Applikationen nutzen ... 948

19 App-Oberflächen entwerfen ... 953

19.1 Grundkonzepte ... 953
19.1.1 XAML (oder HTML 5) für die Oberfläche ... 954
19.1.2 Die Page, der Frame und das Window ... 955
19.1.3 Das Befehlsdesign ... 957
19.1.4 Die Navigationsdesigns ... 959
19.1.5 Achtung: Fingereingabe! ... 960
19.1.6 Verwendung von Schriftarten ... 960
19.2 Seitenauswahl und -navigation ... 961
19.2.1 Die Startseite festlegen ... 961
19.2.2 Navigation und Parameterübergabe ... 961
19.2.3 Den Seitenstatus erhalten ... 962
19.3 App-Darstellung ... 963
19.3.1 Vollbild quer und hochkant ... 963
19.3.2 Was ist mit Andocken und Füllmodus? ... 964
19.3.3 Reagieren auf die Änderung ... 964

19.4 Skalieren von Apps ... 966
19.5 Praxisbeispiele ... 968
19.5.1 Seitennavigation und Parameterübergabe ... 968
19.5.2 Die Fensterkopfzeile anpassen ... 970

20 Die wichtigsten Controls ... 973

20.1 Einfache WinRT-Controls ... 973
20.1.1 TextBlock, RichTextBlock ... 973
20.1.2 Button, HyperlinkButton, RepeatButton ... 976
20.1.3 CheckBox, RadioButton, ToggleButton, ToggleSwitch ... 978
20.1.4 TextBox, PasswordBox, RichEditBox ... 979
20.1.5 Image ... 983
20.1.6 ScrollBar, Slider, ProgressBar, ProgressRing ... 984
20.1.7 Border, Ellipse, Rectangle ... 986
20.2 Layout-Controls ... 987
20.2.1 Canvas ... 987
20.2.2 StackPanel ... 988
20.2.3 ScrollViewer ... 988
20.2.4 Grid ... 989
20.2.5 VariableSizedWrapGrid ... 990
20.2.6 SplitView ... 991
20.2.7 Pivot ... 995
20.2.8 RelativPanel ... 996
20.3 Listendarstellungen ... 998
20.3.1 ComboBox, ListBox ... 998
20.3.2 ListView ... 1002
20.3.3 GridView ... 1004
20.3.4 FlipView ... 1006
20.4 Sonstige Controls ... 1008
20.4.1 CaptureElement ... 1008
20.4.2 MediaElement ... 1009
20.4.3 Frame ... 1011
20.4.4 WebView ... 1011
20.4.5 ToolTip ... 1012
20.4.6 CalendarDatePicker ... 1014
20.4.7 DatePicker/TimePicker ... 1015

20.5 Praxisbeispiele ... 1015
20.5.1 Einen StringFormat-Konverter implementieren ... 1015
20.5.2 Besonderheiten der TextBox kennen lernen ... 1017
20.5.3 Daten in der GridView gruppieren ... 1020
20.5.4 Das SemanticZoom-Control verwenden ... 1025
20.5.5 Die CollectionViewSource verwenden ... 1030
20.5.6 Zusammenspiel ListBox/AppBar ... 1033

21 Apps im Detail ... 1037

21.1 Ein Windows App-Projekt im Detail ... 1037
21.1.1 Contracts und Extensions ... 1038
21.1.2 AssemblyInfo.vb ... 1038
21.1.3 Verweise ... 1040
21.1.4 App.xaml und App.xaml.vb ... 1040
21.1.5 Package.appxmanifest ... 1041
21.1.6 Application1_TemporaryKey.pfx ... 1046
21.1.7 MainPage.xaml & MainPage.xaml.vb ... 1046
21.1.8 Assets/Symbole ... 1047
21.1.9 Nach dem Kompilieren ... 1047
21.2 Der Lebenszyklus einer Windows App ... 1047
21.2.1 Möglichkeiten der Aktivierung von Apps ... 1049
21.2.2 Der Splash Screen ... 1051
21.2.3 Suspending ... 1051
21.2.4 Resuming ... 1052
21.2.5 Beenden von Apps ... 1053
21.2.6 Die Ausnahmen von der Regel ... 1054
21.2.7 Debuggen ... 1054
21.3 Daten speichern und laden ... 1058
21.3.1 Grundsätzliche Überlegungen ... 1058
21.3.2 Worauf und wie kann ich zugreifen? ... 1059
21.3.3 Das AppData-Verzeichnis ... 1059
21.3.4 Das Anwendungs-Installationsverzeichnis ... 1061
21.3.5 Das Downloads-Verzeichnis ... 1062
21.3.6 Sonstige Verzeichnisse ... 1063
21.3.7 Anwendungsdaten lokal sichern und laden ... 1064
21.3.8 Daten in der Cloud ablegen/laden (Roaming) ... 1066
21.3.9 Aufräumen ... 1067
21.3.10 Sensible Informationen speichern ... 1068

21.4 Praxisbeispiele ... 1069
21.4.1 Die Auto-Play-Funktion unterstützen ... 1069
21.4.2 Einen zusätzlichen Splash Screen einsetzen ... 1073
21.4.3 Eine Dateiverknüpfung erstellen ... 1075

22 App-Techniken ... 1081

22.1 Arbeiten mit Dateien/Verzeichnissen ... 1081
22.1.1 Verzeichnisinformationen auflisten ... 1081
22.1.2 Unterverzeichnisse auflisten ... 1084
22.1.3 Verzeichnisse erstellen/löschen ... 1086
22.1.4 Dateien auflisten ... 1087
22.1.5 Dateien erstellen/schreiben/lesen ... 1089
22.1.6 Dateien kopieren/umbenennen/löschen ... 1093
22.1.7 Verwenden der Dateipicker ... 1095
22.1.8 StorageFile-/StorageFolder-Objekte speichern ... 1099
22.1.9 Verwenden der Most Recently Used-Liste ... 1101
22.2 Datenaustausch zwischen Apps/Programmen ... 1102
22.2.1 Zwischenablage ... 1102
22.2.2 Teilen von Inhalten ... 1109
22.2.3 Eine App als Freigabeziel verwenden ... 1112
22.2.4 Zugriff auf die Kontaktliste ... 1113
22.3 Spezielle Oberflächenelemente ... 1115
22.3.1 MessageDialog ... 1115
22.3.2 ContentDialog ... 1118
22.3.3 Popup-Benachrichtigungen ... 1120
22.3.4 PopUp/Flyouts ... 1127
22.3.5 Das PopupMenu einsetzen ... 1131
22.3.6 Eine AppBar verwenden ... 1133
22.4 Datenbanken und Windows Store Apps ... 1137
22.4.1 Der Retter in der Not: SQLite! ... 1137
22.4.2 Verwendung/Kurzüberblick ... 1137
22.4.3 Installation ... 1139
22.4.4 Wie kommen wir zu einer neuen Datenbank? ... 1140
22.4.5 Wie werden die Daten manipuliert? ... 1144
22.5 Vertrieb der App ... 1145
22.5.1 Verpacken der App ... 1145
22.5.2 App-Installation per Skript ... 1147

22.6 Ein Blick auf die App-Schwachstellen 1149
22.6.1 Quellcodes im Installationsverzeichnis 1149
22.6.2 Zugriff auf den App-Datenordner 1151
22.7 Praxisbeispiele 1151
22.7.1 Ein Verzeichnis auf Änderungen überwachen 1151
22.7.2 Eine App als Freigabeziel verwenden 1154
22.7.3 ToastNotifications einfach erzeugen 1159

Anhang

A Glossar 1167

B Wichtige Dateiextensions 1173

Index 1175

Download-Kapitel

LINK: http://doko-buch.de

Vorwort zu den Download-Kapiteln 1215

Teil IV: WPF-Anwendungen

23 Einführung in WPF 1219

23.1 Einführung 1220
23.1.1 Was kann eine WPF-Anwendung? 1220
23.1.2 Die eXtensible Application Markup Language 1222
23.1.3 Verbinden von XAML und Basic-Code 1227
23.1.4 Zielplattformen 1232
23.1.5 Applikationstypen 1233
23.1.6 Vorteile und Nachteile von WPF-Anwendungen 1234
23.1.7 Weitere Dateien im Überblick 1235
23.2 Alles beginnt mit dem Layout 1238
23.2.1 Allgemeines zum Layout 1238
23.2.2 Positionieren von Steuerelementen 1240
23.2.3 Canvas 1243
23.2.4 StackPanel 1244
23.2.5 DockPanel 1246
23.2.6 WrapPanel 1248
23.2.7 UniformGrid 1248
23.2.8 Grid 1250
23.2.9 ViewBox 1254
23.2.10 TextBlock 1255
23.3 Das WPF-Programm 1258
23.3.1 Die Application-Klasse 1259
23.3.2 Das Startobjekt festlegen 1259
23.3.3 Kommandozeilenparameter verarbeiten 1261

23.3.4 Die Anwendung beenden ... 1261
23.3.5 Auswerten von Anwendungsereignissen ... 1262
23.4 Die Window-Klasse ... 1263
23.4.1 Position und Größe festlegen ... 1263
23.4.2 Rahmen und Beschriftung ... 1263
23.4.3 Das Fenster-Icon ändern ... 1264
23.4.4 Anzeige weiterer Fenster ... 1264
23.4.5 Transparenz ... 1264
23.4.6 Abstand zum Inhalt festlegen ... 1265
23.4.7 Fenster ohne Fokus anzeigen ... 1266
23.4.8 Ereignisfolge bei Fenstern ... 1266
23.4.9 Ein paar Worte zur Schriftdarstellung ... 1267
23.4.10 Ein paar Worte zur Controldarstellung ... 1269
23.4.11 Wird mein Fenster komplett mit WPF gerendert? ... 1271

24 Übersicht WPF-Controls ... 1273

24.1 Allgemeingültige Eigenschaften ... 1273
24.2 Label ... 1275
24.3 Button, RepeatButton, ToggleButton ... 1275
24.3.1 Schaltflächen für modale Dialoge ... 1276
24.3.2 Schaltflächen mit Grafik ... 1277
24.4 TextBox, PasswortBox ... 1278
24.4.1 TextBox ... 1278
24.4.2 PasswordBox ... 1280
24.5 CheckBox ... 1281
24.6 RadioButton ... 1282
24.7 ListBox, ComboBox ... 1284
24.7.1 ListBox ... 1284
24.7.2 ComboBox ... 1287
24.7.3 Den Content formatieren ... 1288
24.8 Image ... 1290
24.8.1 Grafik per XAML zuweisen ... 1290
24.8.2 Grafik zur Laufzeit zuweisen ... 1290
24.8.3 Bild aus Datei laden ... 1291
24.8.4 Die Grafikskalierung beeinflussen ... 1292
24.9 MediaElement ... 1293
24.10 Slider, ScrollBar ... 1296
24.10.1 Slider ... 1296

24.10.2 ScrollBar ... 1297
24.11 ScrollViewer ... 1298
24.12 Menu, ContextMenu ... 1299
24.12.1 Menu ... 1299
24.12.2 Tastenkürzel ... 1300
24.12.3 Grafiken ... 1301
24.12.4 Weitere Möglichkeiten ... 1302
24.12.5 ContextMenu ... 1303
24.13 ToolBar ... 1304
24.14 StatusBar, ProgressBar ... 1307
24.14.1 StatusBar ... 1307
24.14.2 ProgressBar ... 1309
24.15 Border, GroupBox, BulletDecorator ... 1310
24.15.1 Border ... 1310
24.15.2 GroupBox ... 1311
24.15.3 BulletDecorator ... 1312
24.16 RichTextBox ... 1314
24.16.1 Verwendung und Anzeige von vordefiniertem Text ... 1314
24.16.2 Neues Dokument zur Laufzeit erzeugen ... 1316
24.16.3 Sichern von Dokumenten ... 1316
24.16.4 Laden von Dokumenten ... 1318
24.16.5 Texte per Code einfügen/modifizieren ... 1319
24.16.6 Texte formatieren ... 1320
24.16.7 EditingCommands ... 1322
24.16.8 Grafiken/Objekte einfügen ... 1322
24.16.9 Rechtschreibkontrolle ... 1324
24.17 FlowDocumentPageViewer & Co. ... 1324
24.17.1 FlowDocumentPageViewer ... 1324
24.17.2 FlowDocumentReader ... 1325
24.17.3 FlowDocumentScrollViewer ... 1325
24.18 FlowDocument ... 1325
24.18.1 FlowDocument per XAML beschreiben ... 1326
24.18.2 FlowDocument per Code erstellen ... 1328
24.19 DocumentViewer ... 1329
24.20 Expander, TabControl ... 1330
24.20.1 Expander ... 1330
24.20.2 TabControl ... 1332
24.21 Popup ... 1333

24.22 TreeView 1335
24.23 ListView 1338
24.24 DataGrid 1339
24.25 Calendar/DatePicker 1339
24.26 InkCanvas 1343
24.26.1 Stift-Parameter definieren 1344
24.26.2 Die Zeichenmodi 1345
24.26.3 Inhalte laden und sichern 1345
24.26.4 Konvertieren in eine Bitmap 1346
24.26.5 Weitere Eigenschaften 1347
24.27 Ellipse, Rectangle, Line und Co. 1347
24.27.1 Ellipse 1347
24.27.2 Rectangle 1348
24.27.3 Line 1348
24.28 Browser 1349
24.29 Ribbon 1351
24.29.1 Allgemeine Grundlagen 1351
24.29.2 Download/Installation 1353
24.29.3 Erste Schritte 1353
24.29.4 Registerkarten und Gruppen 1354
24.29.5 Kontextabhängige Registerkarten 1355
24.29.6 Einfache Beschriftungen 1356
24.29.7 Schaltflächen 1357
24.29.8 Auswahllisten 1358
24.29.9 Optionsauswahl 1361
24.29.10 Texteingaben 1361
24.29.11 Screentips 1362
24.29.12 Symbolleiste für den Schnellzugriff 1363
24.29.13 Das RibbonWindow 1363
24.29.14 Menüs 1364
24.29.15 Anwendungsmenü 1366
24.29.16 Alternativen 1369
24.30 Chart 1369
24.31 WindowsFormsHost 1370

25 Wichtige WPF-Techniken ... 1373

25.1 Eigenschaften ... 1373
25.1.1 Abhängige Eigenschaften (Dependency Properties) ... 1373
25.1.2 Angehängte Eigenschaften (Attached Properties) ... 1374
25.2 Einsatz von Ressourcen ... 1375
25.2.1 Was sind eigentlich Ressourcen? ... 1375
25.2.2 Wo können Ressourcen gespeichert werden? ... 1375
25.2.3 Wie definiere ich eine Ressource? ... 1377
25.2.4 Statische und dynamische Ressourcen ... 1378
25.2.5 Wie werden Ressourcen adressiert? ... 1379
25.2.6 System-Ressourcen einbinden ... 1380
25.3 Das WPF-Ereignis-Modell ... 1380
25.3.1 Einführung ... 1380
25.3.2 Routed Events ... 1381
25.3.3 Direkte Events ... 1383
25.4 Verwendung von Commands ... 1383
25.4.1 Einführung in Commands ... 1384
25.4.2 Verwendung vordefinierter Commands ... 1384
25.4.3 Das Ziel des Commands ... 1386
25.4.4 Vordefinierte Commands ... 1387
25.4.5 Commands an Ereignismethoden binden ... 1387
25.4.6 Wie kann ich ein Command per Code auslösen? ... 1389
25.4.7 Command-Ausführung verhindern ... 1389
25.5 Das WPF-Style-System ... 1389
25.5.1 Übersicht ... 1389
25.5.2 Benannte Styles ... 1390
25.5.3 Typ-Styles ... 1392
25.5.4 Styles anpassen und vererben ... 1393
25.6 Verwenden von Triggern ... 1395
25.6.1 Eigenschaften-Trigger (Property triggers) ... 1395
25.6.2 Ereignis-Trigger ... 1397
25.6.3 Daten-Trigger ... 1398
25.7 Einsatz von Templates ... 1399
25.7.1 Template abrufen und verändern ... 1403
25.8 Transformationen, Animationen, StoryBoards ... 1406
25.8.1 Transformationen ... 1406
25.8.2 Animationen mit dem StoryBoard realisieren ... 1411
25.9 Praxisbeispiel ... 1415

26 WPF-Datenbindung 1419

26.1 Grundprinzip 1419
26.1.1 Bindungsarten 1420
26.1.2 Wann wird eigentlich die Quelle aktualisiert? 1421
26.1.3 Geht es auch etwas langsamer? 1422
26.1.4 Bindung zur Laufzeit realisieren 1423
26.2 Binden an Objekte 1425
26.2.1 Objekte im Code instanziieren 1425
26.2.2 Verwenden der Instanz im VB-Quellcode 1427
26.2.3 Anforderungen an die Quell-Klasse 1427
26.2.4 Instanziieren von Objekten per VB-Code 1429
26.3 Binden von Collections 1430
26.3.1 Anforderung an die Collection 1430
26.3.2 Einfache Anzeige 1431
26.3.3 Navigation zwischen den Objekten 1432
26.3.4 Einfache Anzeige in einer ListBox 1433
26.3.5 DataTemplates zur Anzeigeformatierung 1435
26.3.6 Mehr zu List- und ComboBox 1436
26.3.7 Verwenden der ListView 1438
26.4 Noch einmal zurück zu den Details 1440
26.4.1 Navigieren in den Daten 1440
26.4.2 Sortieren 1441
26.4.3 Filtern 1442
26.4.4 Live Shaping 1443
26.5 Anzeige von Datenbankinhalten 1444
26.5.1 Datenmodell per LINQ to SQL-Designer erzeugen 1444
26.5.2 Die Programm-Oberfläche 1445
26.5.3 Der Zugriff auf die Daten 1447
26.6 Drag & Drop-Datenbindung 1448
26.6.1 Vorgehensweise 1448
26.6.2 Weitere Möglichkeiten 1451
26.7 Formatieren von Werten 1452
26.7.1 IValueConverter 1453
26.7.2 BindingBase.StringFormat-Eigenschaft 1455
26.8 Das DataGrid als Universalwerkzeug 1456
26.8.1 Grundlagen der Anzeige 1457
26.8.2 Vom Betrachten zum Editieren 1461

26.9 Praxisbeispiele ... 1461
26.9.1 Collections in Hintergrundthreads füllen ... 1461
26.9.2 Drag & Drop-Bindung bei 1:n-Beziehungen ... 1465

27 Druckausgabe mit WPF ... 1469

27.1 Grundlagen ... 1469
27.1.1 XPS-Dokumente ... 1469
27.1.2 System.Printing ... 1470
27.1.3 System.Windows.Xps ... 1471
27.2 Einfache Druckausgaben mit dem PrintDialog ... 1471
27.3 Mehrseitige Druckvorschau-Funktion ... 1474
27.3.1 Fix-Dokumente ... 1474
27.3.2 Flow-Dokumente ... 1480
27.4 Druckerinfos, -auswahl, -konfiguration ... 1483
27.4.1 Die installierten Drucker bestimmen ... 1484
27.4.2 Den Standarddrucker bestimmen ... 1485
27.4.3 Mehr über einzelne Drucker erfahren ... 1485
27.4.4 Spezifische Druckeinstellungen vornehmen ... 1487
27.4.5 Direkte Druckausgabe ... 1489

Teil V: Windows Forms

28 Windows Forms-Anwendungen ... 1493

28.1 Grundaufbau/Konzepte ... 1493
28.1.1 Wo ist das Hauptprogramm? ... 1494
28.1.2 Die Oberflächendefinition – Form1.Designer.vb ... 1499
28.1.3 Die Spielwiese des Programmierers – Form1.vb ... 1500
28.1.4 Die Datei AssemblyInfo.vb ... 1501
28.1.5 Resources resx/Resources.Designer.vb ... 1502
28.1.6 Settings.settings/Settings.Designer.vb ... 1503
28.2 Ein Blick auf die Application-Klasse ... 1505
28.2.1 Eigenschaften ... 1505
28.2.2 Methoden ... 1506
28.2.3 Ereignisse ... 1507
28.3 Allgemeine Eigenschaften von Komponenten ... 1508
28.3.1 Font ... 1509
28.3.2 Handle ... 1510

28.3.3 Tag ... 1511
28.3.4 Modifiers ... 1511
28.4 Allgemeine Ereignisse von Komponenten ... 1512
28.4.1 Die Eventhandler-Argumente ... 1512
28.4.2 Sender ... 1512
28.4.3 Der Parameter e ... 1514
28.4.4 Mausereignisse ... 1514
28.4.5 KeyPreview ... 1516
28.4.6 Weitere Ereignisse ... 1517
28.4.7 Validierung ... 1518
28.4.8 SendKeys ... 1518
28.5 Allgemeine Methoden von Komponenten ... 1519

29 Windows Forms-Formulare ... 1521

29.1 Übersicht ... 1521
29.1.1 Wichtige Eigenschaften des Form-Objekts ... 1522
29.1.2 Wichtige Ereignisse des Form-Objekts ... 1524
29.1.3 Wichtige Methoden des Form-Objekts ... 1525
29.2 Praktische Aufgabenstellungen ... 1526
29.2.1 Fenster anzeigen ... 1526
29.2.2 Splash Screens beim Anwendungsstart anzeigen ... 1529
29.2.3 Eine Sicherheitsabfrage vor dem Schließen anzeigen ... 1531
29.2.4 Ein Formular durchsichtig machen ... 1532
29.2.5 Die Tabulatorreihenfolge festlegen ... 1532
29.2.6 Ausrichten und Platzieren von Komponenten ... 1533
29.2.7 Spezielle Panels für flexibles Layout ... 1536
29.2.8 Menüs erzeugen ... 1537
29.3 MDI-Anwendungen ... 1541
29.3.1 "Falsche" MDI-Fenster bzw. Verwenden von Parent ... 1541
29.3.2 Die echten MDI-Fenster ... 1542
29.3.3 Die Kindfenster ... 1543
29.3.4 Automatisches Anordnen der Kindfenster ... 1544
29.3.5 Zugriff auf die geöffneten MDI-Kindfenster ... 1546
29.3.6 Zugriff auf das aktive MDI-Kindfenster ... 1546
29.3.7 Mischen von Kindfenstermenü/MDIContainer-Menü ... 1546
29.4 Praxisbeispiele ... 1547
29.4.1 Informationsaustausch zwischen Formularen ... 1547
29.4.2 Ereigniskette beim Laden/Entladen eines Formulars ... 1555

30 Komponenten-Übersicht ... 1561

30.1 Allgemeine Hinweise ... 1561
30.1.1 Hinzufügen von Komponenten ... 1561
30.1.2 Komponenten zur Laufzeit per Code erzeugen ... 1562
30.2 Allgemeine Steuerelemente ... 1564
30.2.1 Label ... 1564
30.2.2 LinkLabel ... 1565
30.2.3 Button ... 1566
30.2.4 TextBox ... 1567
30.2.5 MaskedTextBox ... 1570
30.2.6 CheckBox ... 1571
30.2.7 RadioButton ... 1573
30.2.8 ListBox ... 1574
30.2.9 CheckedListBox ... 1575
30.2.10 ComboBox ... 1576
30.2.11 PictureBox ... 1577
30.2.12 DateTimePicker ... 1577
30.2.13 MonthCalendar ... 1578
30.2.14 HScrollBar, VScrollBar ... 1578
30.2.15 TrackBar ... 1579
30.2.16 NumericUpDown ... 1580
30.2.17 DomainUpDown ... 1581
30.2.18 ProgressBar ... 1581
30.2.19 RichTextBox ... 1582
30.2.20 ListView ... 1583
30.2.21 TreeView ... 1589
30.2.22 WebBrowser ... 1594
30.3 Container ... 1595
30.3.1 FlowLayout/TableLayout/SplitContainer ... 1595
30.3.2 Panel ... 1595
30.3.3 GroupBox ... 1596
30.3.4 TabControl ... 1597
30.3.5 ImageList ... 1599
30.4 Menüs & Symbolleisten ... 1600
30.4.1 MenuStrip und ContextMenuStrip ... 1600
30.4.2 ToolStrip ... 1600
30.4.3 StatusStrip ... 1600
30.4.4 ToolStripContainer ... 1601

30.5 Daten ... 1601
30.5.1 DataSet ... 1601
30.5.2 DataGridView/DataGrid ... 1602
30.5.3 BindingNavigator/BindingSource ... 1602
30.5.4 Chart ... 1602
30.6 Komponenten ... 1603
30.6.1 ErrorProvider ... 1603
30.6.2 HelpProvider ... 1604
30.6.3 ToolTip ... 1604
30.6.4 Timer ... 1604
30.6.5 BackgroundWorker ... 1604
30.6.6 SerialPort ... 1604
30.7 Drucken ... 1605
30.7.1 PrintPreviewControl ... 1605
30.7.2 PrintDocument ... 1605
30.8 Dialoge ... 1605
30.8.1 OpenFileDialog/SaveFileDialog/FolderBrowserDialog ... 1605
30.8.2 FontDialog/ColorDialog ... 1605
30.9 WPF-Unterstützung mit dem ElementHost ... 1605
30.10 Praxisbeispiele ... 1606
30.10.1 Mit der CheckBox arbeiten ... 1606
30.10.2 Steuerelemente per Code selbst erzeugen ... 1607
30.10.3 Controls-Auflistung im TreeView anzeigen ... 1610
30.10.4 WPF-Komponenten mit dem ElementHost anzeigen ... 1613

31 Grundlagen der Grafikausgabe ... 1617

31.1 Übersicht und erste Schritte ... 1617
31.1.1 GDI+ – Ein erster Blick für Umsteiger ... 1618
31.1.2 Namespaces für die Grafikausgabe ... 1619
31.2 Darstellen von Grafiken ... 1621
31.2.1 Die PictureBox-Komponente ... 1621
31.2.2 Das Image-Objekt ... 1622
31.2.3 Laden von Grafiken zur Laufzeit ... 1623
31.2.4 Sichern von Grafiken ... 1623
31.2.5 Grafikeigenschaften ermitteln ... 1624
31.2.6 Erzeugen von Vorschaugrafiken (Thumbnails) ... 1625
31.2.7 Die Methode RotateFlip ... 1626
31.2.8 Skalieren von Grafiken ... 1627

31.3 Das .NET-Koordinatensystem ... 1628
31.3.1 Globale Koordinaten ... 1629
31.3.2 Seitenkoordinaten (globale Transformation) ... 1630
31.3.3 Gerätekoordinaten (Seitentransformation) ... 1632
31.4 Grundlegende Zeichenfunktionen von GDI+ ... 1633
31.4.1 Das zentrale Graphics-Objekt ... 1633
31.4.2 Punkte zeichnen/abfragen ... 1636
31.4.3 Linien ... 1637
31.4.4 Kantenglättung mit Antialiasing ... 1638
31.4.5 PolyLine ... 1639
31.4.6 Rechtecke ... 1639
31.4.7 Polygone ... 1641
31.4.8 Splines ... 1641
31.4.9 Bézierkurven ... 1643
31.4.10 Kreise und Ellipsen ... 1644
31.4.11 Tortenstück (Segment) ... 1644
31.4.12 Bogenstück ... 1646
31.4.13 Wo sind die Rechtecke mit den "runden Ecken"? ... 1646
31.4.14 Textausgabe ... 1648
31.4.15 Ausgabe von Grafiken ... 1652
31.5 Unser Werkzeugkasten ... 1653
31.5.1 Einfache Objekte ... 1653
31.5.2 Vordefinierte Objekte ... 1654
31.5.3 Farben/Transparenz ... 1656
31.5.4 Stifte (Pen) ... 1658
31.5.5 Pinsel (Brush) ... 1661
31.5.6 SolidBrush ... 1661
31.5.7 HatchBrush ... 1661
31.5.8 TextureBrush ... 1663
31.5.9 LinearGradientBrush ... 1663
31.5.10 PathGradientBrush ... 1665
31.5.11 Fonts ... 1666
31.5.12 Path-Objekt ... 1667
31.5.13 Clipping/Region ... 1670
31.6 Standarddialoge ... 1673
31.6.1 Schriftauswahl ... 1673
31.6.2 Farbauswahl ... 1674

31.7 Praxisbeispiele ... 1676
31.7.1 Ein Graphics-Objekt erzeugen ... 1676
31.7.2 Zeichenoperationen mit der Maus realisieren ... 1679

32 Druckausgabe ... 1683

32.1 Einstieg und Übersicht ... 1683
32.1.1 Nichts geht über ein Beispiel ... 1683
32.1.2 Programmiermodell ... 1685
32.1.3 Kurzübersicht der Objekte ... 1686
32.2 Auswerten der Druckereinstellungen ... 1686
32.2.1 Die vorhandenen Drucker ... 1686
32.2.2 Der Standarddrucker ... 1687
32.2.3 Verfügbare Papierformate/Seitenabmessungen ... 1687
32.2.4 Der eigentliche Druckbereich ... 1689
32.2.5 Die Seitenausrichtung ermitteln ... 1689
32.2.6 Ermitteln der Farbfähigkeit ... 1690
32.2.7 Die Druckauflösung abfragen ... 1690
32.2.8 Ist beidseitiger Druck möglich? ... 1690
32.2.9 Einen "Informationsgerätekontext" erzeugen ... 1691
32.2.10 Abfragen von Werten während des Drucks ... 1692
32.3 Festlegen von Druckereinstellungen ... 1693
32.3.1 Einen Drucker auswählen ... 1693
32.3.2 Drucken in Millimetern ... 1693
32.3.3 Festlegen der Seitenränder ... 1694
32.3.4 Druckjobname ... 1695
32.3.5 Die Anzahl der Kopien festlegen ... 1695
32.3.6 Beidseitiger Druck ... 1696
32.3.7 Seitenzahlen festlegen ... 1697
32.3.8 Druckqualität verändern ... 1700
32.3.9 Ausgabemöglichkeiten des Chart-Controls nutzen ... 1700
32.4 Die Druckdialoge verwenden ... 1701
32.4.1 PrintDialog ... 1701
32.4.2 PageSetupDialog ... 1702
32.4.3 PrintPreviewDialog ... 1704
32.4.4 Ein eigenes Druckvorschau-Fenster realisieren ... 1705
32.5 Drucken mit OLE-Automation ... 1706
32.5.1 Kurzeinstieg in die OLE-Automation ... 1706
32.5.2 Drucken mit Microsoft Word ... 1709

32.6 Praxisbeispiele ... 1712
32.6.1 Den Drucker umfassend konfigurieren ... 1712
32.6.2 Diagramme mit dem Chart-Control drucken ... 1721
32.6.3 Drucken mit Word ... 1723

33 Windows Forms-Datenbindung ... 1729

33.1 Prinzipielle Möglichkeiten ... 1729
33.2 Manuelle Bindung an einfache Datenfelder ... 1729
33.2.1 BindingSource erzeugen ... 1730
33.2.2 Binding-Objekt ... 1730
33.2.3 DataBindings-Collection ... 1731
33.3 Manuelle Bindung an Listen und Tabellen ... 1731
33.3.1 DataGridView ... 1731
33.3.2 Datenbindung von ComboBox und ListBox ... 1732
33.4 Entwurfszeit-Bindung an typisierte DataSets ... 1732
33.5 Drag & Drop-Datenbindung ... 1734
33.6 Navigations- und Bearbeitungsfunktionen ... 1734
33.6.1 Navigieren zwischen den Datensätzen ... 1734
33.6.2 Hinzufügen und Löschen ... 1734
33.6.3 Aktualisieren und Abbrechen ... 1735
33.6.4 Verwendung des BindingNavigators ... 1735
33.7 Die Anzeigedaten formatieren ... 1736
33.8 Praxisbeispiele ... 1736
33.8.1 Einrichten und Verwenden einer Datenquelle ... 1736
33.8.2 Eine Auswahlabfrage im DataGridView anzeigen ... 1740
33.8.3 Master-Detailbeziehungen im DataGrid anzeigen ... 1743
33.8.4 Datenbindung Chart-Control ... 1744

34 Erweiterte Grafikausgabe ... 1749

34.1 Transformieren mit der Matrix-Klasse ... 1749
34.1.1 Übersicht ... 1749
34.1.2 Translation ... 1750
34.1.3 Skalierung ... 1750
34.1.4 Rotation ... 1751
34.1.5 Scherung ... 1751
34.1.6 Zuweisen der Matrix ... 1752
34.2 Low-Level-Grafikmanipulationen ... 1752
34.2.1 Worauf zeigt Scan0? ... 1753

34.2.2 Anzahl der Spalten bestimmen ... 1754
34.2.3 Anzahl der Zeilen bestimmen ... 1755
34.2.4 Zugriff im Detail (erster Versuch) ... 1755
34.2.5 Zugriff im Detail (zweiter Versuch) ... 1757
34.2.6 Invertieren ... 1759
34.2.7 In Graustufen umwandeln ... 1760
34.2.8 Heller/Dunkler ... 1761
34.2.9 Kontrast ... 1762
34.2.10 Gamma-Wert ... 1763
34.2.11 Histogramm spreizen ... 1764
34.2.12 Ein universeller Grafikfilter ... 1766
34.3 Fortgeschrittene Techniken ... 1770
34.3.1 Flackerfrei dank Double Buffering ... 1770
34.3.2 Animationen ... 1772
34.3.3 Animated GIFs ... 1775
34.3.4 Auf einzelne GIF-Frames zugreifen ... 1778
34.3.5 Transparenz realisieren ... 1779
34.3.6 Eine Grafik maskieren ... 1781
34.3.7 JPEG-Qualität beim Sichern bestimmen ... 1782
34.4 Grundlagen der 3D-Vektorgrafik ... 1783
34.4.1 Datentypen für die Verwaltung ... 1784
34.4.2 Eine universelle 3D-Grafik-Klasse ... 1785
34.4.3 Grundlegende Betrachtungen ... 1786
34.4.4 Translation ... 1789
34.4.5 Streckung/Skalierung ... 1789
34.4.6 Rotation ... 1790
34.4.7 Die eigentlichen Zeichenroutinen ... 1792
34.5 Und doch wieder GDI-Funktionen 1795
34.5.1 Am Anfang war das Handle 1795
34.5.2 Gerätekontext (Device Context Types) ... 1797
34.5.3 Koordinatensysteme und Abbildungsmodi ... 1799
34.5.4 Zeichenwerkzeuge/Objekte ... 1804
34.5.5 Bitmaps ... 1806
34.6 Praxisbeispiele ... 1810
34.6.1 Die Transformationsmatrix verstehen ... 1810
34.6.2 Eine 3D-Grafikausgabe in Aktion ... 1813
34.6.3 Einen Fenster-Screenshot erzeugen ... 1816

35 Ressourcen/Lokalisierung ... 1819

35.1 Manifestressourcen ... 1819
35.1.1 Erstellen von Manifestressourcen ... 1819
35.1.2 Zugriff auf Manifestressourcen ... 1821
35.2 Typisierte Resourcen ... 1822
35.2.1 Erzeugen von resources-Dateien ... 1822
35.2.2 Hinzufügen der resources-Datei zum Projekt ... 1823
35.2.3 Zugriff auf die Inhalte von resources-Dateien ... 1823
35.2.4 ResourceManager direkt aus der resources-Datei erzeugen ... 1824
35.2.5 Was sind .resx-Dateien? ... 1825
35.3 Streng typisierte Ressourcen ... 1825
35.3.1 Erzeugen streng typisierter Ressourcen ... 1825
35.3.2 Verwenden streng typisierter Ressourcen ... 1826
35.3.3 Streng typisierte Ressourcen per Reflection auslesen ... 1826
35.4 Anwendungen lokalisieren ... 1828
35.4.1 Localizable und Language ... 1829
35.4.2 Beispiel "Landesfahnen" ... 1829
35.4.3 Einstellen der aktuellen Kultur zur Laufzeit ... 1832
35.5 Praxisbeispiel ... 1833
35.5.1 Betrachter für Manifestressourcen ... 1833

36 Komponentenentwicklung ... 1837

36.1 Überblick ... 1837
36.2 Benutzersteuerelement ... 1838
36.2.1 Entwickeln einer Auswahl-ListBox ... 1838
36.2.2 Komponente verwenden ... 1840
36.3 Benutzerdefiniertes Steuerelement ... 1841
36.3.1 Entwickeln eines BlinkLabels ... 1841
36.3.2 Verwenden der Komponente ... 1843
36.4 Komponentenklasse ... 1843
36.5 Eigenschaften ... 1844
36.5.1 Einfache Eigenschaften ... 1844
36.5.2 Schreib-/Lesezugriff (Get/Set) ... 1845
36.5.3 Nur Lese-Eigenschaft (ReadOnly) ... 1845
36.5.4 Nur-Schreibzugriff (WriteOnly) ... 1846
36.5.5 Hinzufügen von Beschreibungen ... 1846
36.5.6 Ausblenden im Eigenschaftenfenster ... 1847
36.5.7 Einfügen in Kategorien ... 1847

36.5.8 Default-Wert einstellen ... 1848
36.5.9 Standard-Eigenschaft ... 1849
36.5.10 Wertebereichsbeschränkung und Fehlerprüfung ... 1849
36.5.11 Eigenschaften von Aufzählungstypen ... 1851
36.5.12 Standard Objekt-Eigenschaften ... 1852
36.5.13 Eigene Objekt-Eigenschaften ... 1853
36.6 Methoden ... 1855
36.6.1 Konstruktor ... 1855
36.6.2 Class-Konstruktor ... 1857
36.6.3 Destruktor ... 1858
36.6.4 Aufruf des Basisklassen-Konstruktors ... 1858
36.6.5 Aufruf von Basisklassen-Methoden ... 1859
36.7 Ereignisse (Events) ... 1859
36.7.1 Ereignis mit Standardargument definieren ... 1859
36.7.2 Ereignis mit eigenen Argumenten ... 1860
36.7.3 Ein Default-Ereignis festlegen ... 1861
36.7.4 Mit Ereignissen auf Windows-Messages reagieren ... 1862
36.8 Weitere Themen ... 1863
36.8.1 Wohin mit der Komponente? ... 1863
36.8.2 Assembly-Informationen festlegen ... 1865
36.8.3 Assemblies signieren ... 1866
36.8.4 Komponenten-Ressourcen einbetten ... 1867
36.8.5 Der Komponente ein Icon zuordnen ... 1867
36.8.6 Den Designmodus erkennen ... 1868
36.8.7 Komponenten lizenzieren ... 1869
36.9 Praxisbeispiele ... 1873
36.9.1 AnimGif für die Anzeige von Animationen ... 1873
36.9.2 Eine FontComboBox entwickeln ... 1876
36.9.3 Das PropertyGrid verwenden ... 1878

Index ... 1881

Vorwort

Die Zeit, in der Visual Basic-Programmierer meinten, mit ein paar Klicks auf ein Formular und mit wenigen Zeilen Quellcode eine vollständige Applikation erschaffen zu können, ist zumindest seit Anbruch der .NET-Epoche endgültig vorbei. Vorbei ist aber auch die Zeit, in der mancher C-Programmierer mitleidig auf den VB-Kollegen herabblicken konnte. VB ist seit langem zu einem vollwertigen und professionellen Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework geworden, beginnend bei Windows Forms- über WPF-, ASP.NET-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch zu Visual Basic 2015 soll ein faires Angebot sowohl für künftige als auch für fortgeschrittene VB-Programmierer sein. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen siebzehn 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 Visual Basic 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.

Das ist auch der Grund, warum das vorliegende Buch keinen ausgesprochenen Lehrbuchcharakter trägt, sondern mehr ein mit sorgfältig gewählten Beispielen durchsetztes Nachschlagewerk der wichtigsten Elemente der .NET-Programmierung unter Visual Basic 2015 ist.

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 Basic 2015 zu liefern oder all die Informatio-

nen 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 Schneise durch den Urwald der .NET-Programmierung mit Visual Basic 2015 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.

Die Kapitel des Buchs haben wir in fünf Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit VB.NET
2. Technologien
3. Windows Store Apps
4. WPF-Anwendungen
5. Windows Forms-Anwendungen

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 drei Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen zwei Themenkomplexe mussten wir als PDF auslagern, welche Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs und die Kapitel des vierten und fünften Teils können Sie sich unter folgender Adresse herunterladen:

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

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.

- Bei der Fehlermeldung "Der Microsoft.Jet.OLEDB.4.0-Provider ist nicht auf dem lokalen Computer registriert." müssen Sie als Zielplattform für das Projekt x86 wählen, da es sich bei OLEDB um einen 32-Bit-Treiber handelt.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio 2015 unter Windows 8 bzw. 10 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 1900 Seiten – in diesem Buch nicht alles finden, was Visual Basic 2015 bzw. das .NET Framework 4.6 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 August 2015*

Teil I: Grundlagen

- **Einstieg in Visual Studio 2015**
- **Grundlagen der Sprache Visual Basic**
- **Objektorientiertes Programmieren**
- **Arrays, Strings und Funktionen**
- **Weitere wichtige Sprachfeatures**
- **Einführung in LINQ**

Kapitel 1

Einstieg in Visual Studio 2015

Dieses Kapitel bietet dem VB-Programmierer einen effektiven Schnelleinstieg in die Arbeit mit Visual Studio 2015. 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 2015 gegenüber der Vorgängerversion Visual Studio 2012.

1.1 Die Installation von Visual Studio 2015

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

HINWEIS: Voraussetzung für ein erfolgreiches Studium dieses Buchs ist das Vorhandensein eines PCs mit einer lauffähigen Installation von Visual Studio 2015.

1.1.1 Überblick über die Produktpalette

Alle im Handel angebotenen Visual-Studio-Pakete basieren auf dem .NET-Framework 4.6. 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 Community 2015

Bei dieser kostenfreien Version handelt es sich bereits um eine vollständig ausgestattete Entwicklungsumgebung für Windows-Desktop-, Web- und plattformübergreifenden iOS-, Android- und Windows-Applikationen.

Sie kann auch für kommerzielle Projekte eingesetzt werden, wenn es sich dabei um Unternehmen mit weniger als 250 Mitarbeitern handelt.

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

Visual Studio Professional 2015

Wie es der Name bereits suggeriert, handelt es sich bei diesem Standard-Paket bereits um ein professionelles Werkzeug zur Entwicklung beliebiger Anwendungstypen im Team:

- Mit *CodeLens* ist ein leistungsstarkes Features zum Verbessern der Produktivität Ihres Teams enthalten
- Verschiedenen Planungstools (Agile-Projekte, Teamräume, Diagramme, ...) dienen der Verbesserung der Team-Produktivität
- Mit bestimmten MSDN-Abonnementleistungen erhalten Sie Zugang zu nützlicher Software für Entwicklung/Tests, Team Foundation Server, Visual Studio Online Basic ...

Visual Studio Enterprise 2015

Hier handelt es sich um die Vollausstattung für Softwareentwickler, die im Team Anwendungen auf Enterprise-Niveau erstellen wollen. Neben allen Features der Professional-Version sind auch weitere Funktionen enthalten, die eine komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen sollen.

HINWEIS: Visual Studio Enterprise ersetzt die bisherigen Editionen Premium und Ultimate!

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 2012. Die folgende Auflistung kann lediglich eine Orientierungshilfe sein:

- Betriebssystem: Windows 10, 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-Laufwerk

Die Parameter von Prozessor und RAM sind als untere Grenzwerte zu verstehen.

Ganz wichtig:

HINWEIS: Wollen Sie Apps für Windows 8/10 entwickeln, so ist das Betriebssystem Windows 8 bzw. 10 für das Entwicklungssystem unerlässlich!

Weiterhin ist zu beachten:

- Das .NET Framework ab 4.5 wird von Windows XP nicht mehr unterstützt – motten Sie also Ihren alten Computer ein.
- Das .NET-Framework 3.5 ist standardmäßig nicht mehr in Windows 8 bzw. 10 enthalten, es muss nachinstalliert werden (*Systemsteuerung/Programme und Features/Windows Features aktivieren ...*) oder die Anwendungen müssen auf die Version 4 aktualisiert werden.
- Der SQL Server Express ist nicht mehr im Installationspaket enthalten, sondern muss separat heruntergeladen werden. Alternativ steht nach der Installation von Visual Studio der SQL Server Express LocalDB zur Verfügung.

1.2 Unser allererstes VB-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 Visual Basic-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 Visual Studio, kann man 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 VB-Kommandozeilencompiler *vbc.exe*.

Compilerpfad eintragen

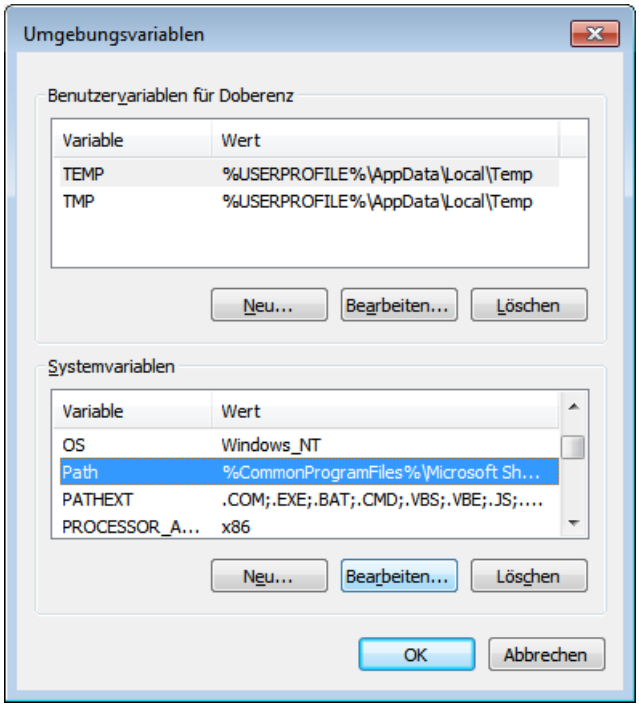
Der VB-Compiler *vbc.exe* befindet sich, ziemlich versteckt, im Verzeichnis

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

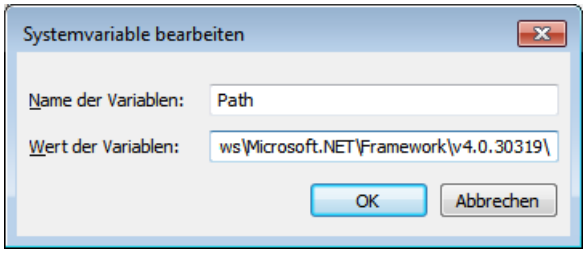
Da das Kompilieren direkt an der Kommandozeile ausgeführt werden soll, werden wir *vbc.exe* in den Windows-Pfad aufnehmen, um so seinen Aufruf von jedem Ordner des Systems aus zu ermöglichen:

- 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 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 *vbc.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:

```
vbc /?
```

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.

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

1.2.2 Programm schreiben

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

```
Imports System
Module KonsolenDemo
    Sub Main()
        Dim kurs, euro, dollar As Single, b As Char
        Console.WriteLine("Umrechnung Euro in Dollar")
        Do
            Console.Write("Kurs 1 : ")
            kurs = CSng(Console.ReadLine())
            Console.Write("Euro: ")
            euro = CSng(Console.ReadLine())
            dollar = euro * kurs
            Console.WriteLine("Sie erhalten " & dollar.ToString("0.00 Dollar"))
            Console.Write("Programm beenden? (j/n)")
            b = CChar(Console.ReadLine())
        Loop While b <> "j"
    End Sub
End Module
```

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

1.2.3 Programm kompilieren und testen

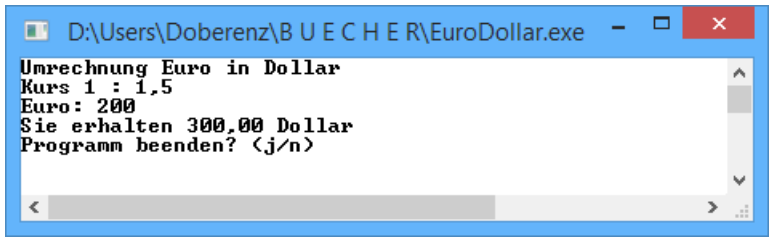
Um bequem an der Kommandozeile arbeiten zu können, kopieren Sie zunächst die Datei *cmd.exe* (Eingabeaufforderung aus *...\Windows\System32*) in dasselbe Verzeichnis, in welchem sich auch die Datei *EuroDollar.vb* befindet.

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

```
vbc /t:exe EuroDollar.vb
```

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 VB-Programm aus!



HINWEIS: Ist die eingegebene Zahl komplett, so ist mittels Enter-Taste abzuschließen!

1.2.4 Einige Erläuterungen zum Quellcode

Da wir auf die Grundlagen der Sprache Visual Basic erst in den späteren Kapiteln ausführlich zu sprechen kommen, sollen einige Vorabinformationen den größten Wissensdurst stillen.

Befehlszeilen

Das Ende einer VB-Befehlszeile wird in der Regel durch einen Zeilenumbruch markiert. Die auszuführenden Befehlszeilen befinden sich innerhalb der *SubMain*-Prozedur, die mit *End Sub* abgeschlossen wird.

Imports-Anweisung

Mit der ersten Anweisung

```
Imports 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.

Noch kürzer wird es mit

```
Imports System.Console
```

denn dann würde die folgende Anweisung genügen:

```
WriteLine("Umrechnung Euro in Dollar")
```

Module-Anweisung

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

HINWEIS: Ein VB-Programm besteht aus mindestens einer **.vb*-Textdatei mit einem Modul (oder 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 VB-Compiler erzeugten Datei *EuroDollar.exe* handelt es sich **nicht** um eine herkömmliche Exe-Anwendung, 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).

1.2.5 Konsolenanwendungen sind out

Zwar hat seit Visual Basic 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 – außer nostalgischen Erinnerungen an die DOS-Steinzeit – noch positive Emotionen?

Als einfaches Hilfsmittel zum Erlernen von VB und für verschiedenste Testzwecke, hat dieser einfache Anwendungstyp aber durchaus noch seine Daseinsberechtigung.

Mit Visual Studio 2015 werden wir im Praxisteil dieses Kapitels (Abschnitt 1.7.2) das gleiche Problem lösen, diesmal allerdings mit einer attraktiven Windows-Oberfläche. Bevor es aber so richtig losgehen kann, sollten wir uns zunächst ein wenig mit der Windows-Philosophie anfreunden.

1.3 Die Windows-Philosophie

Eine moderne Programmiersprache wie Visual Basic 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 einfache 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 VB mit folgenden Objekten zu tun:

- **Formulare** Das sind die Fenster, in welchen Ihre VB-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* (Event-Handler).

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 Windows-Programmierung unter Visual Studio 2015

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

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

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 notwendig) in ihrer Größe verändert.

Bereits während der ersten Etappe hat man – mehr oder weniger unbewusst – Eigenschaften verändert, wie z.B.Position und Abmessungen von Formularen und Steuerelementen. 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* (Event-Handlern) 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) für alle zum jeweiligen Objekt passenden Ereignisse zur Verfügung. Der Programmierer füllt diesen Rahmen mit VB-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 VB-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 2015* zu einer hocheffektiven Entwicklungsumgebung für Windows- und Web-Anwendungen.

1.4 Die Entwicklungsumgebung Visual Studio 2015

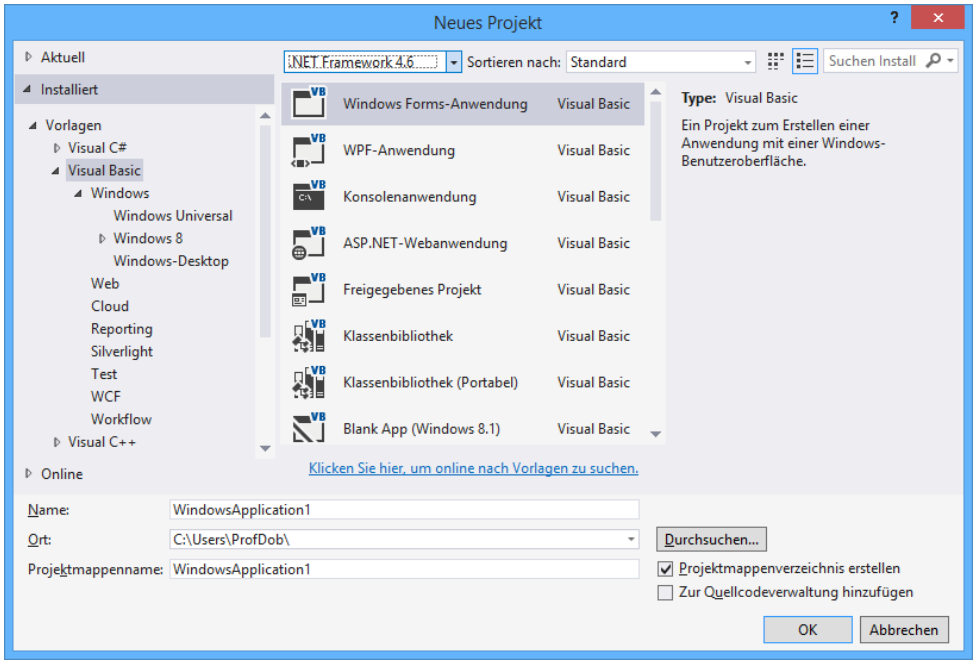
Visual Studio 2015 ist eine universelle Entwicklungsumgebung (IDE[1]) für Windows- und für Web-Anwendungen, die auf Microsofts .NET-Technologie basieren. Alle notwendigen Tools, wie z.B. für den visuellen Oberflächenentwurf, für die Codeprogrammierung und für die Fehlersuche, werden bereitgestellt.

Visual Basic ist nur eine der möglichen objektorientierten Sprachen, die Sie unter Visual Studio 2015 einsetzen können. So werden z.B. noch Visual C#, Visual C++ und Visual F# unterstützt.

HINWEIS: Die folgenden kurzen Erklärungen sollen 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 2015 den Menüpunkt *Neues Projekt...*, so öffnet sich der Startdialog *Neues Projekt*. Unter der Rubrik *Andere Sprachen* (links) werden Sie mit einem umfangreichen und zunächst verwirrenden Angebot an unterschiedlichen Vorlagen[2] für Visual Basic-Projekttypen konfrontiert, wobei für den Einsteiger zunächst die klassische *Windows Forms-Anwendung* empfohlen wird.



[1] *Integrated Developers Environment*

[2] Die Abbildung bezieht sich auf die Community-Edition.

Wie Sie es an der linken oberen Klappbox sehen, kann man Programme für verschiedene .NET-Framework-Versionen entwickeln (Multi-Targeting).

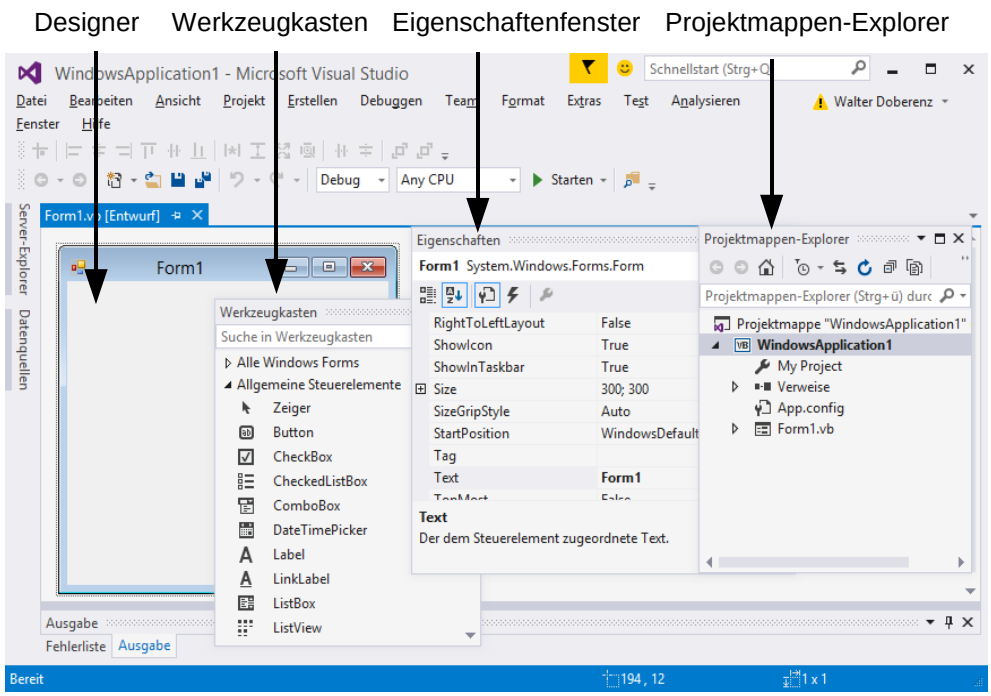
Haben Sie das Häkchen bei *Projektmappenverzeichnis erstellen* gesetzt, so erzeugt Visual Studio automatisch einen Unterordner mit dem Namen des Projekts in dem als Speicherort eingetragenen Hauptverzeichnis.

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 etwa den folgenden Anblick bieten, wobei auf die für den Einsteiger zunächst wichtigsten Fenster besonders hingewiesen wird.

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



Der Projektmappen-Explorer

Da es unter Visual Studio 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 auch als "Schaltzentrale" Ihres Projekts betrachten.

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

Zur Bedeutung der einzelnen Einträge:

- *My Project*
  Hier sind verschiedene Dateien zusammengefasst, die die Projekteigenschaften bestimmen. *AssemblyInfo.vb* enthält z.B allgemeine Infos zur Assemblierung des Projekts, wie Titel, Beschreibung, Versionsnummer, Copyright. Weitere Dateien beziehen sich auf die Ressourcen und die Projekteinstellungen.
- *Form1.vb*
  Diese Datei enthält eine partielle Klasse, die den von Ihnen selbst hinzugefügten Code von *Form1* kapselt.

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

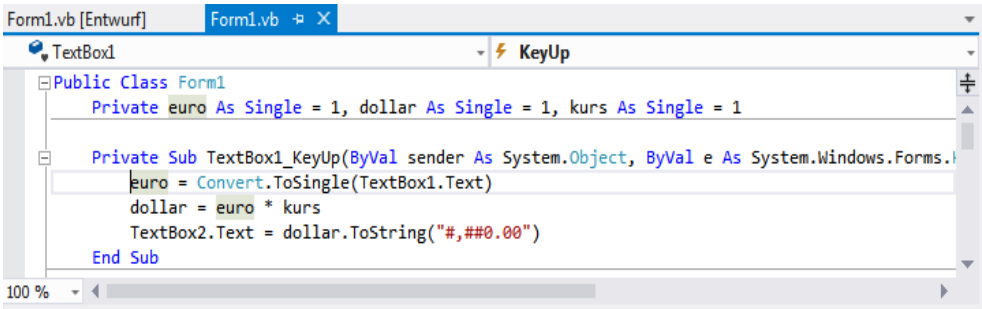
Im Eigenschaften-Fenster (Menü *Ansicht/Eigenschaftenfenster* bzw. *F4*) werden die zur Entwurfszeit editierbaren Eigenschaften des gerade aktiven Steuerelements aufgelistet[1]. Normalerweise hat jede Eigenschaft bereits einen Standardwert, den Sie in vielen Fällen übernehmen können.

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] Lassen Sie sich nicht davon irritieren, dass das Eigenschaftenfenster nicht nur die Eigenschaften, sondern 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 VB-Programmierer. Um zum Beispiel das zu *Form1* (siehe obige Abbildung) gehörige Codefenster zu öffnen, klicken Sie auf den Designer von *Form1* mit der rechten Maustaste und wählen im Kontextmenü *Code anzeigen* (F7). Die folgende Abbildung zeigt das Codefenster für *Form1* des zweiten Praxisbeispiels am Kapitelende.



Sie öffnen das Codefenster, indem Sie z.B. im Projektmappen-Explorer oder im Designer auf einen Eintrag bzw. ein Objekt mit der rechten Maustaste klicken und im Kontextmenü *Code anzeigen* wählen (F7).

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 wir 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 sollen Sie sich mit der grundlegenden .NET-Philosophie und den damit verbundenen Konzepten, Begriffen und Features anfreunden. 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 in den letzten Jahren dramatisch verändert haben, wobei das Internet mit seinen hohen Ansprüchen an die Skalierbarkeit einer Anwendung, die Verteilbar-

keit auf mehrere Schichten und ausreichende Sicherheit der hauptsächliche Motor war, sich nach einer grundlegend neuen Sprachkonzeption umzuschauen.

Vom alten VB zu VB.NET

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.

Als konsequent objektorientierte Sprache erfüllt VB.NET 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).

Die einst hoch gelobte COM[1]-Technologie wurde zum Auslaufmodell erklärt. Microsoft kann natürlich nicht diese Technologie auf die Müllkippe entsorgen, denn zu viele Programmierer würden dadurch verprellt werden. Aus diesem Grund wird COM auch in .NET noch lange Zeit sein Gnadenbrot erhalten.

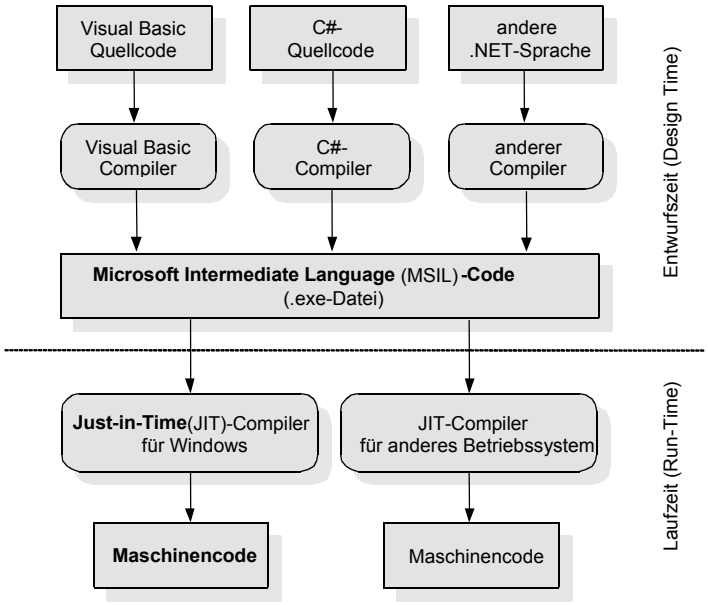
Wie funktioniert eine .NET-Sprache?

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. Ein .NET-Programm wird also vom Entwurf bis zu seiner Ausführung auf dem Zielrechner tatsächlich zweimal kompiliert (siehe folgende Abbildung).

[1] *Component Object Model*



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.

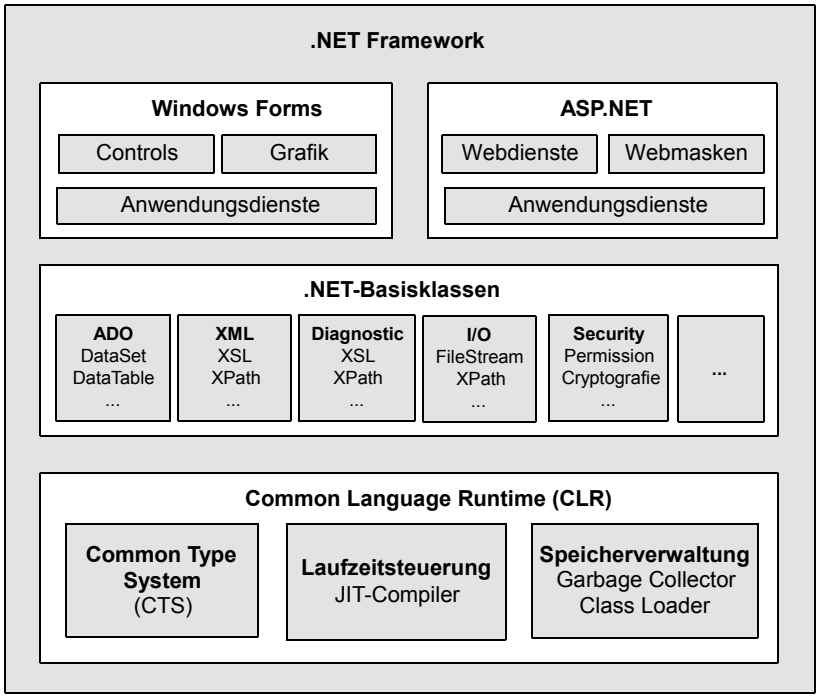
1.5.2 .NET-Features und Begriffe

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.



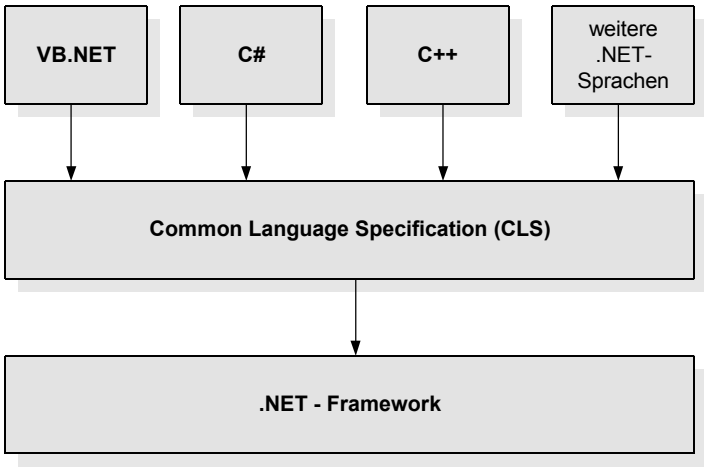
Zu den wichtigsten Komponenten des .NET-Frameworks und den damit zusammenhängenden Begriffen zählen:

- 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 VB, C# ...

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.

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.

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

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.

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 Delphi.NET 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. Anstatt 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 den Einbau der zusätzlichen Interoperabilitätsschicht sinkt die Performance meist deutlich ab.

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 auch eine Liste 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 der .NET-Reflector.

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

Attribute

Wer noch in älteren objektorientierten Sprachen (z.B. VB 6, Delphi 7) zu Hause ist, 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 pur

Last, but not least wollen wir am Ende unserer kleinen Rundreise durch 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 Visual Basic fortzufahren ohne sich vorher mit dem Konzept der OOP vertraut gemacht zu haben (siehe Kapitel 3).

1.6 Wichtige Neuigkeiten in Visual Studio 2015

Für alle Leser, die bereits mit einer Vorgängerversion gearbeitet haben, dürften die folgenden Ausführungen von Interesse sein. Neueinsteiger können diesen Abschnitt überspringen.

1.6.1 Entwicklungsumgebung

Hier möchten wir vor allem auf zwei Neuerungen verweisen:

- Zum Erstellen von leeren, freigegebenen Projekten gibt es neue Vorlagen. Auf diese freigegebenen Projekte kann jetzt durch verschiedene andere Projekttypen verwiesen werden (Konsolenanwendungen, Klassenbibliotheken, Windows Forms-Apps, Windows Universal Apps, Windows Store 8.1, Windows Phone 8.1, ...).
- Benutzerdefinierte Fensterlayouts lassen sich jetzt speichern, indem Sie im Menü *Fenster* auf *Fensterlayout speichern* klicken. Auch das Löschen, Umbenennen und Neuanordnen von Layouts ist möglich (Menü *Fenster / Fensterlayout verwalten*).
- Der Debugger hat ein neues Design und außerdem einige zusätzliche Funktionalitäten (Lambda-Support, Watch-Fenster).

1.6.2 Neue VB-Sprachfeatures

Die Liste der Neuerungen fällt relativ bescheiden aus und beschränkt sich auf einige Spezialfälle:

- Das Beschreiben von Formatstrings lässt sich mittels Stringinterpolation vereinfachen
- Nullbedingte Operatoren wurden eingeführt
- Kommentare sind jetzt auch in Fortsetzungszeilen möglich
- Zeichenkettenliterale sind auch über mehrere Zeilen erlaubt
- Um den Namen eines Parameters, Members oder Typs als Zeichenfolge zu gewinnen, gibt es mit der *NameOf*-Funktion jetzt einen typsicheren Weg
- Es gibt auch selbst implementierende Read-Only Eigenschaften
- Nicht nur Formulare und Klassen, sondern jetzt auch Module und Interfaces können *Partial* deklariert werden

1.6.3 Code-Editor

Erst auf den zweiten Blick sieht man, dass die Benutzeroberfläche des Code-Editors etwas aufpoliert wurde:

- Das Glühbirnen-Symbol vereinfacht einige Aktionen, wie die Beseitigung von Tippfehlern und das Umgestalten von Code. Es werden Vorschläge zur Problemlösung angezeigt.

- Beim Umbenennen werden jetzt alle Instanzen des umzubenennenden Bezeichners hervorgehoben, der neue Namen des Bezeichners braucht nur einmal im Editor eingegeben zu werden
- Die Auswertung von Ausdrücken wurde verbessert. Auch LINQ- und Lambda-Ausdrücke werden jetzt im Überwachungs- und im Direktfenster unterstützt.

1.6.4 NET Framework 4.6

Zu dieser neuesten Version von .NET Framework wurden einige neue APIs hinzugefügt, um vor allem plattformübergreifende Szenarien zu ermöglichen:

- Neue Kryptografie-APIs, wie etwa *AsymmetricAlgorithm.KeyExchangeAlgorithm*, *AsymmetricAlgorithm.SignatureAlgorithm*
- Das Feature[1] "Ändern der Größe von Windows Forms-Steuerelementen" umfasst jetzt auch die *System.Windows.Forms*-Typen *DomainUpDown*, *NumericUpDown*, *DataGridViewComboBoxColumn*, *DataGridViewColumn* und *ToolStripSplitButton*
- Ein *EventSource*-Objekt kann jetzt direkt konstruiert werden, Sie können eine der *Write()*-Methoden aufrufen, um ein sich selbst beschreibendes Ereignis abzugeben

1.7 Praxisbeispiele

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 alles 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 2015 zu gewinnen.

1.7.1 Windows-Anwendung für Einsteiger

Die bescheidene Funktionalität beschränkt sich auf ein Fensterchen mit einer Schaltfläche, über welche per Mausklick die Beschriftung der Titelleiste in "Hallo VB-Freunde" geändert werden kann. Das Beispiel demonstriert, mit welch geringem Aufwand man in Visual Studio eigene 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 2015* erfolgt entweder über das Windows-Startmenü oder schneller über eine vorher eingerichtete Desktop- bzw. Taskleisten-Verknüpfung.

[1] Zur Aktivierung müssen Sie *EnableWindowsFormsHighDpiAutoResizing* in der *app.config* auf *True* setzen.

Auf der Startseite klicken Sie den Link *Neues Projekt...*. Im sich daraufhin öffnenden Dialogfenster *Neues Projekt* wählen Sie links in der Baumstruktur unter dem Knoten *Vorlagen/Andere Sprachen/Visual Basic* zunächst *Windows* aus (siehe Abschnitt 1.4.1).

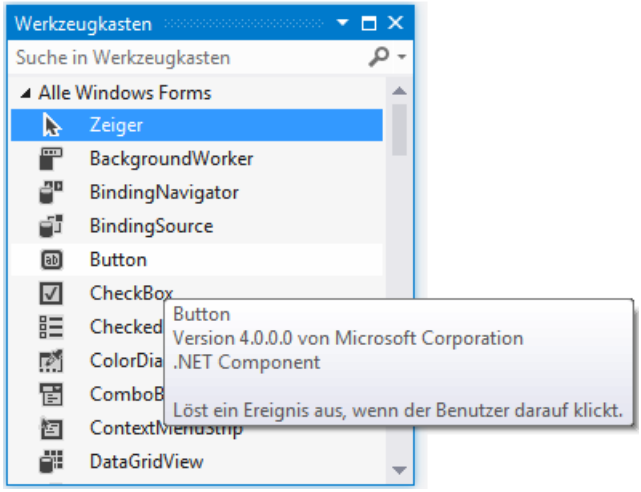
Im Mittelteil klicken Sie auf *Windows Forms-Anwendung* (ganz oben in der Liste). Nehmen Sie im unteren Teil die folgenden Einträge vor bzw. belassen es bei den Standardvorgaben:

Name:	*WindowsApplication1*
Ort:	z.B.: *C:\VB\Beispiele*
Projektmappenname:	*WindowsApplication1*

HINWEIS: Im Dialogfenster rechts oben sehen Sie, dass Sie mit Visual Studio 2015 sowohl Projekte für das .NET Framework 4.6 als auch für die Vorgängerversionen 4.5.2, 4.5.1. 4.5, 3.0 und sogar 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 die gewünschte Komponente aus.

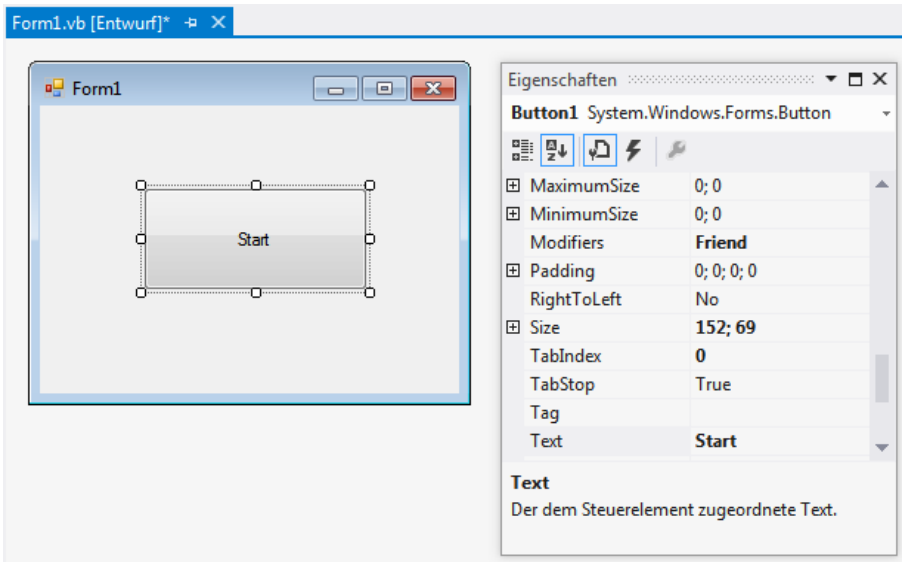


Ein schneller Doppelklick befördert das Steuerelement direkt auf das Formular. Sie können aber auch den gewünschten *Button* erst mit einem einfachen Mausklick im Werkzeugkasten aktivieren, um ihn anschließend auf dem Formular abzusetzen und auf die gewünschte Größe zu zoomen.

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 Objekt 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".

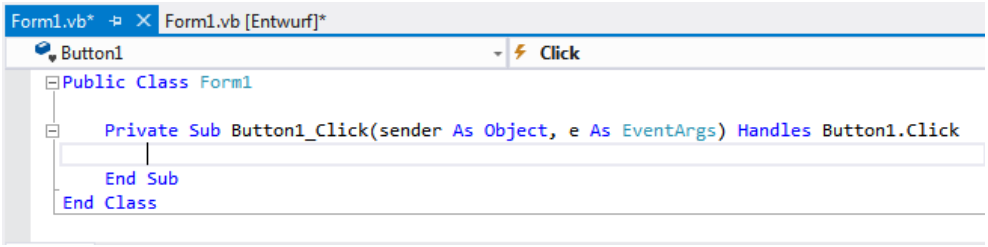
HINWEIS: Verwechseln Sie die *Text*-Eigenschaft nicht mit der *Name*-Eigenschaft. Wenn Sie ein neues Steuerelement platzieren, setzt Visual Studio standardmäßig den Wert von *Text* zunächst auf den von *Name*.



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 Ihr Augenmerk auf die Schreibmarke, welche im vorgefertigten Rahmencode für die *Click*-Ereignisbehandlungsroutine (Event-Handler) blinkt. Hier tragen Sie Ihren VB-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.

Fügen Sie dazu die fett hervorgehobene Anweisung in den Rahmencode ein:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Me.Text = "Hallo VB - Freunde!"
End Sub
```

4. Etappe: Programm kompilieren und testen

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

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

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

Bemerkungen

- Bei diesem Prográmmchen haben Sie ganz nebenbei auch gelernt, dass man 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 (*Me*) durch einen Punkt getrennt.
- Die Properties, die Sie im Eigenschaften-Fenster zuweisen, bezeichnet man auch als *Starteigenschaften*. Zur Laufzeit können diese – wie im Beispiel für die *Text*-Eigenschaft des Formulars gezeigt – durchaus ihren Wert ändern.
- Die als Ergebnis des Kompilierprozesses generierte *.exe*-Datei finden Sie, ziemlich versteckt, im Unterverzeichnis *...\WindowsApplikation1\bin\Debug* des Projektordners. Es handelt sich hierbei allerdings **nicht** um eine klassische Exe-Datei, sondern um eine so genannte *Assemblierung* (siehe Abschnitt 1.5.2). Da diese Exe im MSIL-Code vorliegt, ist sie nur auf solchen Rechnern lauffähig, auf denen vorher die Laufzeitumgebung (Runtime) des .NET-Frameworks installiert wurde. Wenn Sie die Programmentwicklung abgeschlossen und Visual Studio beendet haben, so können Sie später jederzeit in dieses Verzeichnis wechseln, um durch Doppelklick auf die Datei *WindowsApplication1.exe* das fertige Programm zur Ausführung zu bringen.
- Direkt im Projektverzeichnis befindet sich die Projektmappendatei *WindowsApplication1.sln*. Wenn Sie auf diese Datei doppelklicken, so wird standardmäßig Visual Studio geöffnet und das komplette Programm in die Entwicklungsumgebung geladen.
- Falls nach Doppelklick auf die Projektmappendatei **.sln* zwar Visual Studio startet, die Entwicklungsumgebung aber leer bleibt, sollten Sie zunächst den Projektmappen-Explorer öffnen (Menü *Ansicht/Projektmappen-Explorer*) und dann durch Doppelklick (z.B. auf *Form1.vb*) die einzelnen Fenster in die Entwurfsansicht bringen. Zur Codeansicht wechseln Sie entweder mit F7 oder über das Kontextmenü (rechte Maustaste).

1.7.2 Windows-Anwendung für fortgeschrittene Einsteiger

Diesmal soll es keine Spielerei mehr sein, sondern ein durchaus nützliches Prográmmchen – die Umrechnung von Euro in Dollar, also ein simpler Währungsrechner. Durch Vergleichen mit der von uns bereits in 1.2 geschriebenen ersten VB-Anwendung dürften auch die Unterschiede der klassischen Konsolentechnik zur visualisierten, objekt- und ereignisorientierten Windows-Programmierung deutlich werden.

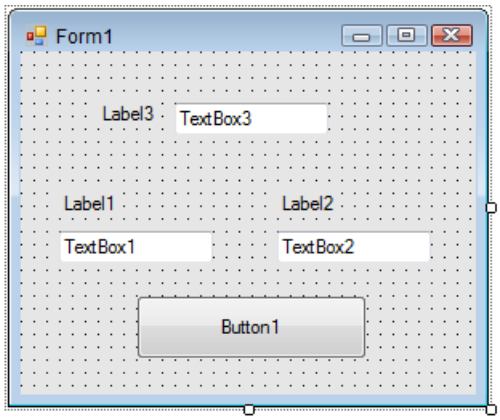
1. Etappe: Visueller Entwurf der Bedienoberfläche

Öffnen Sie ein neues Visual Basic-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)[1].

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 Steuerelemente, nicht auf deren Beschriftung, da die 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 *Label*-Komponente. Der Mauszeiger wechselt sein Aussehen.
- Danach bewegen Sie den Mauszeiger zu der Stelle von *Form1*, an welcher sich die linke obere Ecke von *Label1* befinden soll, drücken die Maustaste nieder und zoomen (bei gedrückt gehaltener Maustaste) das Label auf seine endgültige Größe. Analog verfahren Sie mit *Label2* und *Label3*.
- Nun klicken Sie im Werkzeugkasten auf das Symbol für die *TextBox*-Komponente und erzeugen nacheinander *TextBox1*, *TextBox2* und *TextBox3*.
- Schließlich bleibt noch *Button1*, den Sie am unteren Rand von *Form1* absetzen.

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

2. Etappe: Zuweisen der Objekteigenschaften

Unser Progrämmchen 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:

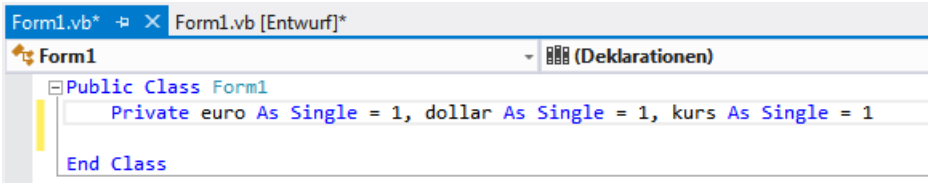
Name des Objekts	Eigenschaft	Neuer Wert
Form1	*Text* *Font.Size*	Umrechnung Euro-Dollar *10*
Label1	*Text*	Euro
Label2	*Text*	Dollar
Label3	*Text*	Kurs 1:
TextBox1	*TextAlign*	*Right*
TextBox2	*TextAlign*	*Right*
TextBox3	*TextAlign*	*Center*
Button1	*Text*	Beenden

3. Etappe: Verknüpfen der Objekte mit Ereignissen

Während Sie die beiden Vorgängeretappen noch getrost Ihrer Sekretärin überlassen konnten, beginnt jetzt Ihre Hauptarbeit als VB-Programmierer. Wechseln Sie zum Code-Fenster *Form1.vb* (auch mit *F7*, *Ansicht*|*Code* oder dem Kontextmenü des Formulars möglich). Was Sie hier erwartet, ist die von Visual Studio vorbereitete Klassendeklaration von *Form1*.

Zunächst fügen Sie eine Anweisung ein, mit der drei Gleitkommavariablen des *Single*-Datentyps deklariert werden. Gleichzeitig werden diese Variablen mit dem Wert 1 initialisiert:

```
Private euro As Single = 1, dollar As Single = 1, kurs As Single = 1
```



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

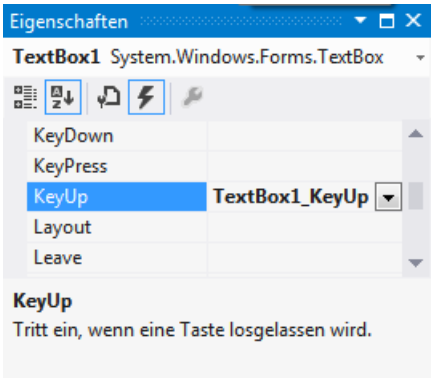
Dabei ist eine fast schon rituelle Erstellungsreihenfolge zu beachten, wie Sie sie mit fortschreitender Programmierpraxis sehr bald auch im Schlaf ausführen können:

- **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 Event-Handlers generiert und im Code-Fenster angezeigt.

- **Ereignisbehandlungen programmieren**
  Füllen Sie den Event-Handler mit den gewünschten VB-Anweisungen aus.

Wir beginnen in unserem Beispiel mit dem *KeyUp*-Event-Handler für *TextBox1*, der immer dann ausgeführt wird, wenn der Benutzer den Euro-Betrag ändert:



Füllen Sie nun den Event-Handler wie folgt aus:

```
Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        euro = Convert.ToSingle(TextBox1.Text)
```

```
        dollar = euro * kurs
        TextBox2.Text = dollar.ToString("#,##0.00")
End Sub
```

HINWEIS: Sie müssen nur die hier fett gedruckten Anweisungen selbst einfügen, der übrige Rahmencode wird automatisch erzeugt, wenn Sie die oben erläuterte Erstellungsreihenfolge beachten!

Der Prozedurkopf des Event-Handlers verweist standardmäßig vor dem Unterstrich (_) auf den Namen des Objekts (*TextBox1*) und danach auf das entsprechende Ereignis (*KeyUp*). Das vorangestellte *Private* verdeutlicht, dass auf die Prozedur nur innerhalb des *Form1*-Klasse zugegriffen werden kann.

Auf analoge Weise erzeugen Sie die Event-Handler für die Steuerelemente *TextBox2* und *TextBox3*.

Ändern des Dollar-Betrags:

```
Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        dollar = Convert.ToSingle(TextBox2.Text)
        euro = dollar / kurs
        TextBox1.Text = euro.ToString("#,##0.00")
End Sub
```

Ändern des Umrechnungskurses:

```
Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        kurs = Convert.ToSingle(TextBox3.Text)
        dollar = euro * kurs
        TextBox2.Text = dollar.ToString("#,##0.00")
End Sub
```

HINWEIS: Denken Sie jetzt noch nicht über den tieferen Sinn der einzelnen Anweisungen nach, denn dazu sind die späteren Kapitel da.

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

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        TextBox1.Text = euro.ToString
        TextBox2.Text = dollar.ToString
        TextBox3.Text = kurs.ToString
End Sub
```

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:

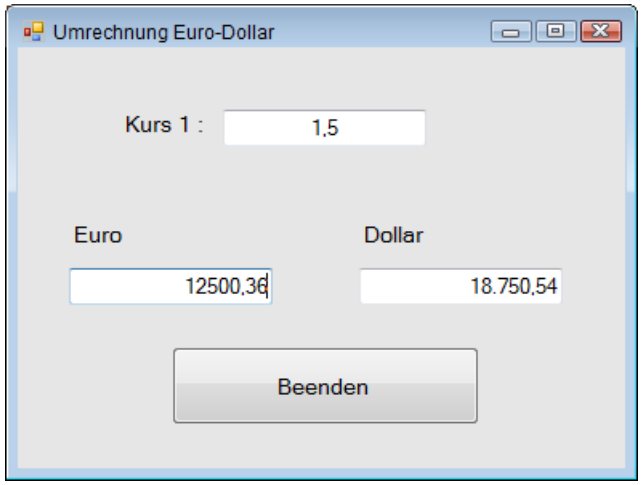
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Me.Close()
End Sub
```

4. Etappe: Programm kompilieren und testen

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

Spielen Sie ruhig ein wenig mit verschiedenen Werten herum, um sich den Unterschied zwischen Konsolen- und Windows-Programmen so richtig 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 nicht erforderlich.

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

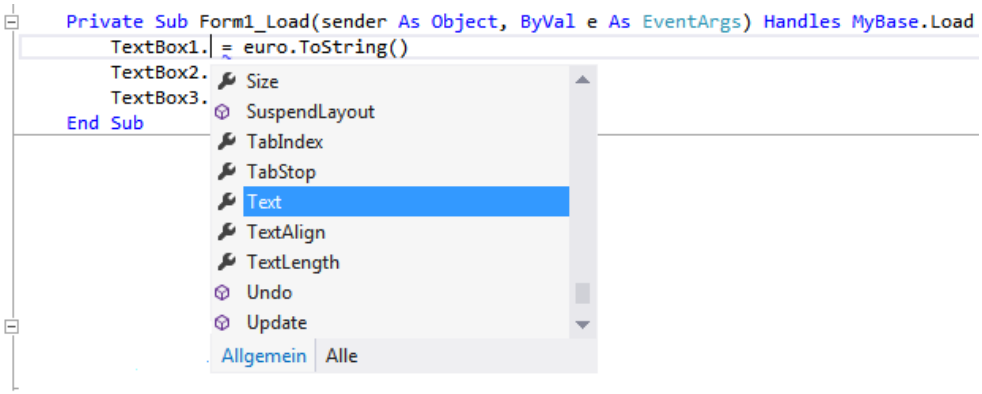


Da wir aus Gründen der Einfachheit in diesem Programm auf eine Überprüfung der Eingabewerte (Eingabevalidierung) verzichtet haben, sollten Sie auf das komplette Löschen des Inhalts eines Textfelds verzichten und nur gültige Zahlenwerte eingeben (also z.B. auch keine Buchstaben). Andernfalls ist ein Konvertieren der Eingabe in eine Zahl nicht möglich und das Programm stürzt mit einer umfangreichen Fehlermeldung ab.

Um wieder in den Entwurfsmodus zurückzukehren, wählen Sie nach solch einem Fehler das Menü *Debuggen/Debugging beenden*.

IntelliSense – die hilfreiche Fee

Eines der bemerkenswertesten Features des Code-Editors von Visual Studio ist die 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 vom lästigen Nachschlagen in der Hilfe und bewahrt Sie vor Schreibfehlern.

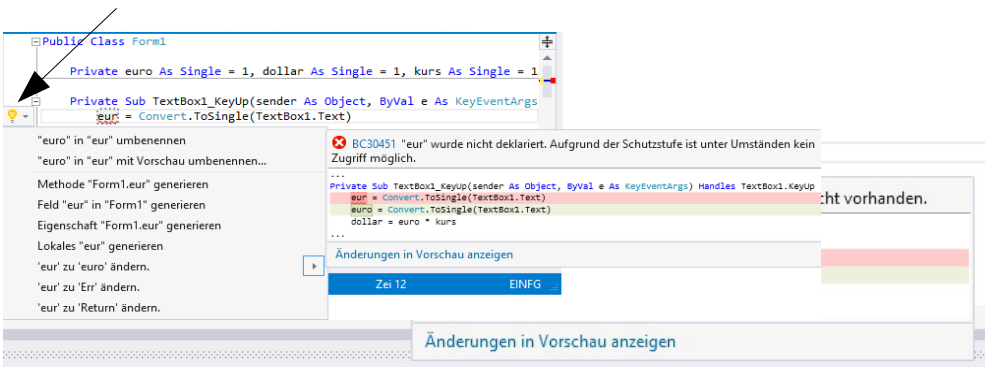


HINWEIS: Wenn Sie das markierte Element übernehmen wollen, brauchen Sie den Namen nicht zu Ende zu schreiben, da die IntelliSense den kompletten Text automatisch ergänzt.

Die Glühbirne sorgt für Erleuchtung

Bereits beim Schreiben des Quellcodes werden Sie von Visual Studio auf grundsätzliche Syntaxfehler (z.B. ein vergessenes Komma) hingewiesen, im Allgemeinen geschieht dies durch Unterstreichen mit einer wellenförmigen (roten) Linie. Wenn Sie mit der Maus auf diese Linie zeigen, erscheint links unterhalb das Symbol einer gelben Glühbirne[1] mit Hinweisen zur Behebung des Fehlers.

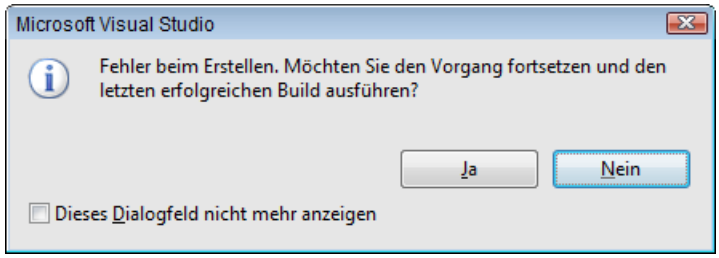
In der folgenden Abbildung wurde z.B. die Variable *euro* falsch eingetippt:



Fehlersuche

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

[1] Dieses Feature gehört zu den Neuigkeiten des Code-Editors von Visual Studio 2015.



Normalerweise 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 mit einer Wellenlinie unterstrichen sind. Auch hier erhalten Sie Hinweise zur Fehlerursache, wenn Sie mit der Maus auf die betreffende Stelle zeigen.

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 (Menü *Ansicht*|*Fehlerliste*).

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

Kapitel 2

Einführung in Visual Basic

In diesem Kapitel wollen wir Ihnen den für den Einstieg wichtigen Sprachkern von Visual Basic erklären. 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 Sprachfeatures erst in späteren Kapiteln (objektorientiertes Programmieren in Kapitel 3, fortgeschrittene Sprachelemente in den Kapiteln 4 und 5).

HINWEIS: Testen Sie möglichst viele der kleinen Codeschnipsel selbst am eigenen PC. Als Codegerüst könnte entweder eine Konsolenanwendung (siehe dazu das Einführungskapitel, Abschnitt 1.2) oder aber eine Windows Forms-Anwendung (siehe 1.7.1/ 1.7.2) dienen.

2.1 Grundbegriffe

Zur Vorbereitung empfehlen wir ein Rückblättern zu den Praxisbeispielen des Kapitels 1, wo einige grundlegende Begriffe (Anweisungen, Klassen, Namensraum, Gültigkeitsbereiche) bereits grob erklärt wurden.

2.1.1 Anweisungen

Wir verstehen unter einer Anweisung (Statement) einen Befehl (Instruction), der sich aus bestimmten Sprachelementen, wie Schlüsselwörtern, Operatoren, Variablen, Konstanten, Ausdrücken, zusammensetzt.

Wir unterscheiden zwei Arten von Anweisungen:

- Deklarations-Anweisungen,
  diese spezifizieren eine Variable, Konstante, Prozedur oder einen Datentyp.
- Ausführbare Anweisungen,
  diese führen bestimmte Aktionen aus (Aufruf einer Prozedur, Zuweisen eines Werts, Durchlaufen einer Schleife etc.).

Wie in jeder anderen Programmiersprache auch, müssen Visual Basic-Anweisungen bestimmten Regeln entsprechen, die man in ihrer Gesamtheit als *Syntax*[1] bezeichnet.

Durch gute Strukturierung Ihres Quellcodes, wie z.B. blockweises Einrücken, machen Sie Ihre Programme übersichtlicher und verringern somit die Fehlerquote. Normalerweise brauchen Sie sich aber darum nicht selbst zu kümmern, denn das erledigt die IDE automatisch für Sie. Anderenfalls gilt:

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 (Variablen, Methoden, Klassen, ...) verwendet man so genannte Bezeichner.

Namensgebung und Schlüsselwörter

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 und ohne eingeschlossene Punkte und Typkennzeichen auskommen.
- Ein Bezeichner muss innerhalb seines Gültigkeitsbereichs eindeutig sein (also keine Mehrfachvergabe des gleichen Namens).
- Bezeichner müssen kürzer als 256 Zeichen sein und
- dürfen keine *Schlüsselwörter* sein.

Bei *Schlüsselwörtern* handelt es sich um vordefinierte reservierte Bezeichner, die den Kern von Visual Basic ausmachen, z.B. *Class, Dim*, *Sub, If, End* etc.

HINWEIS: Schlüsselwörter sind im Codefenster von Visual Studio standardmäßig blau eingefärbt.

BEISPIEL 2.1: Zulässige und unzulässige Namensgebung

VB
```
euro
_radius
Zwerg7
```

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

BEISPIEL 2.1: Zulässige und unzulässige Namensgebung

```
%Anteil                    ' unzulässig, weil "%"-Zeichen am Anfang
7Zwerge                    ' unzulässig, weil Ziffer am Anfang
dim                        ' unzulässig, weil Schlüsselwort
```

Leerzeichen etc.

Leerzeichen zwischen den Befehlswörtern sind im Allgemeinen bedeutungslos und werden in der Regel von Visual Studio automatisch korrigiert. Außerdem gibt es eine ganze Reihe weiterer Zeichen, die z.B. als Operatoren fungieren.

BEISPIEL 2.2: Berechnung eines Ausdrucks mit Divisions- und Additions-Operator

VB
```
z = x/y + 10
```

Groß-/Kleinschreibung

Da die Groß- und Kleinschreibung in Visual Basic egal ist, dürfen Sie keine Bezeichner verwenden, die sich lediglich durch die Groß- und Kleinschreibung voneinander unterscheiden. Sie sollten die Groß-/Kleinschreibung aber dafür einsetzen, um die Lesbarkeit Ihres Programms zu verbessern.

BEISPIEL 2.3: Beide Bezeichner dürfen Sie nicht nebeneinander verwenden. Der erste Bezeichner ist besser lesbar als der zweite und sollte verwendet werden.

VB
```
MeineAdresse       ' gut lesbar
meineadresse       ' doppelter Bezeichner, schlecht lesbar
```

2.1.3 Kommentare

Kommentaranweisungen (im Editor im Allgemeinen grün hervorgehoben) dienen als zusätzliche Erläuterungen für den Programmierer und werden vom Compiler überlesen. Für das Kennzeichnen von Kommentaren verwenden Sie den einfachen Apostroph (')[1].

HINWEIS: Im Unterschied zu anderen Sprachen kennt Visual Basic keine mehrzeiligen Kommentare.

BEISPIEL 2.4: Kommentare

VB
```
x = y + 10                 ' Addiere 10 zum Wert von y
' Jede neue Kommentarzeile muss extra eingeleitet werden.
```

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".

[1] Gelegentlich findet man auch noch das veraltete REM.

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.

Seit VB 2015 sind Kommentare auch bei impliziter Zeilenfortführung erlaubt.

BEISPIEL 2.5: Kommentare bei impliziter Zeilenfortführung

VB
```
Dim mitarbeiter =  {"Müller",    ' Gruppenleiter
                    "Schulze",   ' Stellvertreter
                    "Lehman"}
```

2.1.4 Zeilenumbruch

In früheren Versionen von Visual Basic stand normalerweise pro Zeile eine Anweisung, d h., ein Zeilenumbruch war in der Regel gleichbedeutend mit dem Anweisungsende, es sei denn, man benutzte einen Unterstrich als Zeilentrenner. Mit der Version 2010 wurde diese "eiserne Regel" total aufgeweicht. Bevor wir aber auf die zahlreichen Ausnahmen von dieser Regel eingehen, wollen wir uns mit der klassischen Vorgehensweise beschäftigen, die nach wie vor gültig ist.

Lange Zeilen teilen

Wenn eine Anweisung sehr lang ist, kann das ihre Lesbarkeit im Quelltexteditor beeinträchtigen, und es ist außerdem unbequem, da man horizontal scrollen muss. Man kann aber eine solche Zeile durch einen Unterstrich mit vorangestelltem Leerzeichen trennen.

BEISPIEL 2.6: Die erste Zeile eines Event-Handlers wird auf zwei Zeilen aufgeteilt.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) _
                                                    Handles Button1.Click
```

HINWEIS: Zeichenketten müssen Sie vorher in Teilketten zerlegen, bevor Sie sie trennen!

BEISPIEL 2.7: Eine Zeichenkette wird korrekt getrennt.

VB
```
Label1.Text = "Dies ist eine sehr lange Zeichenkette, die sehr unübersichtlich wäre " &
              ", wenn man sie nicht trennen würde."
```

Implizite Zeilenfortsetzung

In folgenden Situationen kann man eine Anweisung auch ohne Verwendung eines Unterstrichs über mehrere Zeilen verteilen:

- nach einem Komma
- nach öffnenden oder vor schließenden Klammern (einfache oder geschweifte)

- nach dem Zeichen "&"
- nach Zuweisungs- und Binäroperatoren (z.B. =, +=, +, -, Mod, Or, ...)
- nach dem Punkt (.) als Objektqualifizierer
- innerhalb von LINQ-Ausdrücken
- in einigen weiteren Spezialfällen (nach *In* in *For Each*-Schleifen, nach *Is* bzw. *IsNot* ...)
- an bestimmten Stellen im XML-Code, z.B. nach dem Öffnen eines eingebetteten Ausdrucks (<%=) oder vor dem Schließen (%>)

Obige Aufzählung ist nicht ganz vollständig, im Zweifelsfall sollte man den Unterstrich verwenden oder einfach ausprobieren, ob die mehrzeilige Anweisung von der IDE angenommen bzw. richtig verarbeitet wird.

BEISPIEL 2.8: Die Kopfzeile des Eventhandlers im obigen Beispiel kann auch ohne Unterstrich auf zwei Zeilen verteilt werden

VB
```
Private Sub Button1_Click(sender As Object,
                          e As EventArgs) Handles Button1.Click
```

BEISPIEL 2.9: Eine weitere alternative Schreibweise für das Vorgängerbeispiel

VB
```
Private Sub Button1_Click(
                             sender As Object,
                             e As EventArgs
                         ) _
                         Handles Button1.Click
```

BEISPIEL 2.10: Eine LINQ-Abfrage (siehe Kapitel 6) sieht jetzt schön und übersichtlich aus

VB
```
Dim expr = From z In zahlen
           Where z > 10
           Order By z
           Select z
```

Mehrere Anweisungen pro Zeile

Um Platz zu sparen, können aber auch mehrere Anweisungen pro Zeile geschrieben werden, diese sind dann durch einen Doppelpunkt (:) voneinander zu trennen.

BEISPIEL 2.11: Die Anweisungen

VB
```
euro = CSng(TextBox1.Text)
dollar = euro * kurs
```

kann man wie folgt in eine Zeile schreiben:

```
euro = CSng(TextBox1.Text) : dollar = euro * kurs
```

2.2 Datentypen, Variablen und Konstanten

Jedes Programm "lebt" in erster Linie von seinen Variablen und Konstanten, die bestimmten Datentypen entsprechen.

2.2.1 Fundamentale Typen

Die folgende Tabelle gibt eine Übersicht der einfachen (fundamentalen) Datentypen, die Visual Basic zur Verfügung stellt[1]:

VB-Datentyp	.NET-CLR-Typ	Erläuterung	Länge [Byte]
Short	*System.Int16*	kurze Ganzzahl zwischen -2^{15} ... $2^{15}-1$ (-32.768 ... 32.767)	2
Integer	*System.Int32*	Ganzzahl zwischen -2^{31} ... $2^{31}-1$ (-2.147.483.648 ... 2.147.483.647)	4
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
Single	*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
Date	*System.DateTime*	Datum/Zeit zwischen 1. Januar 1 bis 31. Dezember 9999	8
Char	*System.Char*	ein beliebiges Unicode-Zeichen	2
String	*System.String*	beliebige Unicode-Zeichenfolge mit einer maximalen Länge von ca. 2 000 000 000 Zeichen	2 pro Zeichen plus 10
Boolean	*System.Boolean*	Wahrheitswert (*True*, *False*)	2
Byte	*System.Byte*	positive Ganzzahl zwischen 0 ... 255	1
Object	*System.Object*	universeller Datentyp	4

Wie Sie obiger Tabelle entnehmen, entsprechen alle Visual Basic-Datentypen einer Klassendefinition im .NET-Framework.

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

Die CLR-Datentypen sind im *System*-Namensraum angeordnet. Bei der Deklaration (siehe Abschnitt 2.2.4) ist es egal, welchen der beiden möglichen Typbezeichner Sie angeben.

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 VB.

Wertetypen

Dazu zählen all die einfachen Datentypen wie *Byte*, *Integer*, *Double* ..., hinzu kommen später noch andere wie beispielsweise *Structure* (siehe 2.7.2 und *DateTime* (siehe Seite 302). 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

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 möglichst mit einem Kleinbuchstaben.
- Falls Bezeichner aus mehreren Wörtern zusammengesetzt sind, so sollten ab dem zweiten Wort alle Wörter mit einem Großbuchstaben beginnen ("Kamelschreibweise").

BEISPIEL 2.12: Ein Variablenname, der sich aus mehreren Wörtern zusammensetzt.

VB
```
meinHaushaltskassenSaldo
```

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

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

HINWEIS: Die Konventionen der Namensgebung von Variablen gelten auch für Konstanten, Funktionen/Prozeduren und andere benutzerdefinierte Sprachelemente.

2.2.4 Deklaration von Variablen

Variablen sind benannte Speicherplatzstellen, der Variablenname dient dazu, die Speicheradresse im Programmcode anzusprechen.

Dim-Anweisung

Sie können Ihre Variablen mit dem Schlüsselwort *Dim* deklarieren und ihnen mit *As* einen Datentyp zuweisen[1].

SYNTAX: `Dim` *`Varaiablenname`* `As` *`Datentyp`*

BEISPIEL 2.13: Verschiedene Variablendeklarationen

VB
```
Dim a As Single
Dim anzahl As Integer
Dim breite As Double
```

In einer Anweisung lassen sich auch mehrere Variablen hintereinander deklarieren.

BEISPIEL 2.14: Die gleichen Deklarationen wie im Vorgängerbeispiel

VB
```
Dim a As Single, anzahl As Integer, breite As Double
```

Laut obiger Tabelle kann man als Datentyp auch den CLR-Typ angeben.

BEISPIEL 2.15: Die folgenden Deklarationen sind gleichwertig.

VB
```
Dim i As Integer
Dim i As System.Int32
Dim i As Int32
```

Mehrfachdeklaration

Wenn Sie mehrere Variablen vom gleichen Datentyp hintereinander deklarieren wollen, genügt es, den Datentyp nur einmal anzugeben.

BEISPIEL 2.16: Mehrfachdeklaration

VB
Statt

```
Dim i As Integer
Dim j As Integer
```

[1] Später werden Sie erfahren, dass man außer mit *Dim* auch noch mit *Private*, *Public* oder *Static* deklarieren kann.

BEISPIEL 2.16: Mehrfachdeklaration

dürfen Sie auch schreiben

```
Dim i, j As Integer
```

Anfangswerte

Zusammen mit der Deklaration können Sie die Variablen auch gleich initialisieren.

BEISPIEL 2.17: Anfangswerte

VB

Statt

```
Dim max As Single
max = 99.99
```

können Sie kürzer formulieren:

```
Dim max As Single = 99.99
```

Option Explicit

Dieser Schalter, der (für Sie zunächst unsichtbar) am Beginn eines Moduls steht

SYNTAX: `Option Explicit On|Off`

erzwingt die Deklaration einer Variablen, bevor sie verwendet wird (*On*), oder ermöglicht die sofortige Verwendung ohne vorherige Deklaration (*Off*).

BEISPIEL 2.18: Sie haben vergessen, die Variable *j* zu deklarieren

VB

```
Option Explicit On
Dim i As Integer
j = 10        ' undeklarierte Variable erzeugt Compiler-Fehler !
i = 10
```

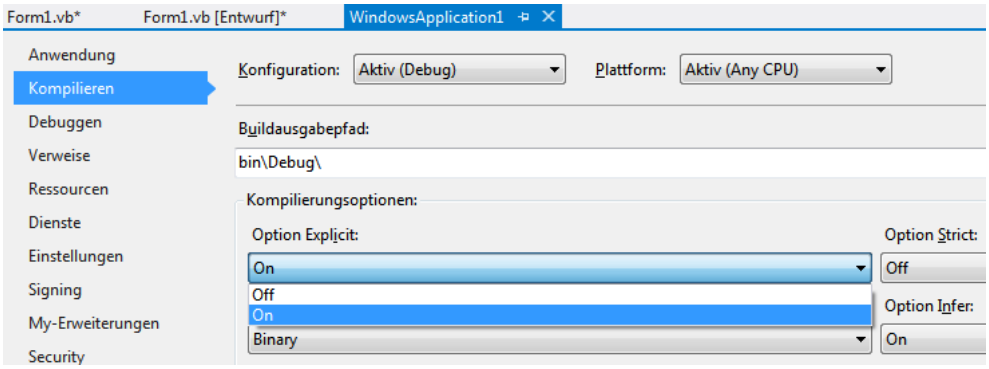
... aber so bemerken Sie Ihren Fehler nicht:

```
Option Explicit Off
Dim i As Integer
j = 10        ' neue Variable vom Object-Datentyp wird erzeugt !
i = 10
```

HINWEIS: Sie sollten grundsätzlich mit *Option Explicit On* arbeiten, da dies die Typsicherheit erhöht und die Fehlerquote senkt.

Glücklicherweise ist *Option Explicit On* die Standardeinstellung. Wenn Sie im Projektmappen-Explorer auf den Projektnamen rechtsklicken und *Eigenschaften* wählen, können Sie im Projekt-

eigenschaftenfenster unter "Kompilieren" die Kompilierungsoption *Option Explicit* für die gesamte Anwendung neu festlegen.



Option Strict

Ähnlich wie *Option Explicit* steht dieser Schalter am Beginn eines Moduls oder kann ebenfalls im Projekteigenschaftenfenster eingestellt werden (siehe vorhergehender Abschnitt).

SYNTAX: `Option Strict On|Off`

Ist die Schalterstellung *Off* (unter Visual Studio 2015 ist das der Standard), so gibt es einige (fragwürdige) Erleichterungen beim Zuweisen unterschiedlicher Datentypen. So kann z.B. bei der Variablendeklaration auf die Angabe des Datentyps verzichtet werden, die Variable ist dann automatisch vom Typ *Object*.

BEISPIEL 2.19: Der folgende Code ergibt einen Fehler, da die *As*-Klausel fehlt:

VB
```
Option Strict On
...
Dim x
```

Dieser Code aber lässt sich fehlerfrei kompilieren:

```
Option Strict Off

...
Dim x         ' x ist vom Object-Datentyp
```

HINWEIS: Der Lernende ist gut beraten, wenn er zunächst mit *Option Strict On* arbeitet, da dies das Verständnis der Programmiersprache fördert und das Portieren des Codes nach anderen .NET-Sprachen (z.B. C#) deutlich erleichtert!

2.2.5 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.

Die Deklaration erfolgt mit dem Schlüsselwort *Dim*, wobei man auf die *As*-Klausel verzichtet.

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

Manch ein VB-Entwickler wird denken, dass es sich hierbei um dasselbe Verhalten wie bei *Option Strict Off* handelt, tatsächlich aber erhalten Sie eine streng typisierte Variable.

BEISPIEL 2.20: Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

VB
```
Dim a = 35
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
Dim a As Integer = 35
```

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

BEISPIEL 2.21: Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *Double*-Wert zugewiesen werden.

VB
```
Dim b = 7
b = 12.3           ' Fehler!
```

HINWEIS: Typinferenz funktioniert nur bei lokalen Variablen!

2.2.6 Konstanten deklarieren

Im Unterschied zu Variablen bleibt der Wert einer Konstanten während der gesamten Laufzeit eines Programms unverändert. Sie legen ihn einmalig mit der *Const*-Anweisung fest. Es ist allgemein üblich, den Namen einer Konstanten in Großbuchstaben zu schreiben.

BEISPIEL 2.22: Konstanten

VB
```
Const PI As Single = 3.1415
Const MELDUNG As String = "Achtung"
Const ANZAHL As Integer = 5
Const X1 As Double = 3 / 4, x2 As Double = 2 + 1 / 3
```

Sammlungen von Konstanten werden üblicherweise in so genannten *Enumerationen* "zusammengehalten" (siehe 2.7.1).

2.2.7 Gültigkeitsbereiche von Deklarationen

Bis jetzt hatten wir Variablen ausschließlich mit dem Zugriffsmodifizierer *Dim* deklariert. Es gibt allerdings noch mindestens drei weitere Alternativen (*Static*, *Private*, *Public*), die sich bezüglich *Sichtbarkeit* und *Lebensdauer* unterscheiden. Was bedeutet das?

- Sichtbarkeit
  ... bestimmt, von welchen Stellen des Programms auf die Variable zugegriffen werden darf.
- Lebensdauer
  ... beschreibt, wie lange die Variable den für sie reservierten Speicherplatz beansprucht.

Die folgende Tabelle vermittelt dazu zunächst einen allgemeinen Überblick:

Zugriffs-modifizierer	Deklaration in	Sichtbarkeit	Lebensdauer
Dim	Sub/Funktion	lokal	bis Prozedur verlassen wird
Static	Sub/Funktion	lokal statisch	Programmlaufzeit
Private	Modul	modulglobal	Programmlaufzeit
Public	Modul	programmglobal	Programmlaufzeit

Nun wollen wir einen genaueren Blick auf die einzelnen Deklarationsmöglichkeiten werfen.

2.2.8 Lokale Variablen mit Dim

Eine lokale Variable wird in der Regel mit *Dim* deklariert und ist nur innerhalb der Prozedur[1] (genauer genommen innerhalb des Blocks) bekannt, in welcher sie deklariert wurde. Das bedeutet, dass sich ihre Sichtbarkeit auf diese Prozedur beschränkt und von außerhalb der Lese- und Schreibzugriff verwehrt ist. Bei jedem neuen Aufruf der Prozedur wird die Variable mit ihrem Leerwert (0 oder Leerstring"") neu initialisiert. Beim Verlassen der Prozedur werden alle mit *Dim* deklarierten Variablen zerstört, und der von ihnen belegte Speicherplatz wird wieder freigegeben.

BEISPIEL 2.23: Platzieren Sie auf einem Formular einen Button und hinterlegen Sie dessen *Click*-Ereignis wie folgt.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
Dim anzahl As Integer       ' lokale Variable
  anzahl = anzahl + 1
  MessageBox.Show(anzahl.ToString)
End Sub
```

[1] *Prozedur* ist der Überbegriff für *Subroutinen* (Sub) und *Funktionen* (Function), siehe 2.8.

Bei jedem Klick auf den Button zeigt das Meldungsfenster "1" an, obwohl wir eigentlich "hochzählen" wollten. Eine Lösung bietet das Deklarieren mit *Static* (siehe nächster Abschnitt).

Die Gültigkeit lokaler Variablen innerhalb einer Prozedur kann weiter eingegrenzt werden, wenn Sie diese innerhalb eines Anweisungsblocks deklarieren. Allerdings darf dann im übergeordneten Block eine gleichnamige Variable nicht nochmals auftreten.

BEISPIEL 2.24: Die Variable *s* gilt nur innerhalb des *If-Then*-Blocks.

VB
```
Dim i As Integer = 5
If i < 5 Then Dim s As String = "Ich bin in einem Block deklariert!"
MessageBox.Show(s)         ' Fehler: Variable s unbekannt
```

2.2.9 Lokale Variablen mit Static

Eine Deklaration mit *Static* anstatt *Dim* bedeutet, dass der Wert einer Variablen nicht "verlorengeht", wenn die Prozedur, in welcher die Variable deklariert wurde, zwischenzeitlich verlassen und später wieder aufgerufen wird.

BEISPIEL 2.25: Bei jedem Klick auf den *Button* erhöht sich der angezeigte Wert um eins.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
  Static anzahl As Integer     ' lokale Variable
  anzahl = anzahl + 1
  MessageBox.Show(anzahl.ToString)
End Sub
```

HINWEIS: *Static*-Deklarationen können **nicht** für globale Variablen angewendet werden!

2.2.10 Private globale Variablen

Mit *Private* (oder ausnahmsweise auch *Dim*) deklarieren Sie globale Variablen, deren Sichtbarkeit sich auf das Modul beschränkt. Eine solche Variable kann von jeder Prozedur innerhalb des Moduls aufgerufen und modifiziert werden. Außerhalb des Moduls ist die Variable unbekannt. Logischerweise deklarieren Sie eine private globale Variable außerhalb jeglicher Prozedur, d h., im Allgemeinteil eines Moduls.

HINWEIS: *Dim* auf Modulebene bedeutet so viel wie *Private*, sollte aber im Interesse eines sauberen Programmierstils nicht verwendet werden.

BEISPIEL 2.26: Im Einführungsbeispiel 1.7.2 hatten wir die drei privaten globalen Variablen *euro*, *dollar* und *kurs* deklariert und mit Anfangswerten initialisiert:

VB
```
Private euro As Single =1, dollar As Single = 1, kurs As Single =1
```

2.2.11 Public Variablen

Wenn Sie möchten, dass auf Modulebene deklarierte Variablen auch noch allen anderen Modulen Ihrer Anwendung zur Verfügung stehen sollen, dann müssen Sie diese mit *Public* deklarieren.

BEISPIEL 2.27: Der Name einer Datenbankdatei, die Sie in Ihrem Projekt von mehreren Formularen aus öffnen müssen, wird in einer anwendungsglobalen Variablen gespeichert.

VB
```
Public dbName As String = "C:\Beispiele\Datenbankanwendung1\Firma.mdb"
```

2.3 Wichtige Datentypen im Überblick

In diesem Abschnitt wollen wir die fundamentalen Datentypen (siehe Tabelle Seite 92) etwas genauer unter die Lupe nehmen und auf einige Besonderheiten aufmerksam machen.

2.3.1 Byte, Short, Integer, Long

Die Datentypen *Byte* (8 Bit), *Short* (16 Bit), *Integer* (32 Bit) und *Long* (64 Bit) dienen dem Speichern von ganzzahligen Werten. Bei der Auswahl ist darauf zu achten, ob der Datentyp für Ihre Zwecke groß genug ist und ob evtl. ein Vorzeichen benötigt (welches *Byte* nicht bietet) wird.

Wenn Sie die Werte als Literale – d.h. direkt im Quellcode – zuweisen, müssen Sie natürlich auch auf die Einhaltung der Wertebereiche achten.

BEISPIEL 2.28: Der folgende Code führt zu einer Fehlermeldung: "Der Konstantenausdruck ist im Typ "Short" nicht repräsentierbar."

VB
```
Dim i As Short
i = 100000            ' Fehler
```

HINWEIS: Bei eingeschalteter strenger Typprüfung müssen die zugewiesenen Literale dem Datentyp der Variablen entsprechen.

BEISPIEL 2.29: Der folgende Code führt zur Fehlermeldung "Option Strict On lässt keine impliziten Typkonvertierungen von Double in Short zu."

VB
```
Dim i As Short
i = 10.5              ' Fehler bei Option Strict On oder 10  bei Option Strict Off
```

Wenn Sie im Beispiel aber *Option Strict Off* einstellen, so wird die letzte Anweisung ohne Murren akzeptiert, und *i* erhält den gerundeten Wert (10).

Für ein ganzzahliges Literal können auch die hexadezimale (Präfix **&H**) oder oktale Schreibweise (Präfix **&O**) Verwendung finden.

BEISPIEL 2.30: Die drei folgenden Zuweisungen sind identisch.

VB
```
Dim b As Byte
b = 255
b = &HFF
b = &O377
```

2.3.2 Single, Double und Decimal

Der *Single*-Typ genügt nur recht bescheidenen Ansprüchen, denn bei einer Vorkommastelle kann er nur etwa sieben Nachkommastellen speichern, während es bei *Double* 15 sind. Der hochgenaue *Decimal*-Datentyp hat 29 Stellen, die sich auf Vor- und Nachkomma aufteilen.

HINWEIS: Wenn Sie Gleitkommazahlen im Quelltext zuweisen, dürfen Sie nicht das Komma, sondern müssen den Punkt als Dezimaltrenner verwenden.

BEISPIEL 2.31: Deklarieren und Zuweisen einer Gleitkommavariablen

VB
```
Dim a As Single
a = 0.45
```

Andererseits dürfen Sie sich nicht wundern, wenn z.B. bei Zahleneingaben in ein Textfeld nur das Komma als Dezimaltrennzeichen akzeptiert wird. Dies hängt mit der deutschen Ländereinstellung zusammen (Windows-Systemsteuerung).

2.3.3 Char und String

Beide Datentypen basieren auf dem Unicode-Zeichensatz, der pro Zeichen 2 Byte beansprucht (im Unterschied zu dem 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.

Den Typ *Char* verwenden Sie zum Speichern eines einzelnen Zeichens, während Zeichenketten im *String*-Datentyp gespeichert werden. Haben Sie die strenge Typprüfung eingeschaltet, so sind *Char*-Literale mit dem Literalzeichen *c* abzuschließen.

BEISPIEL 2.32: Einer *Char*-Variablen wird das Zeichen "A" zugewiesen.

VB
```
Dim z As Char = "A"c
```

Stringausdrücke schließt man in "Gänsefüßchen" ein.

```
Dim s As String = "Ich bin ein String!"
```

Einzelne Strings kann man mit dem "+"- oder "&"-Operator zusammenfügen. Letztere Variante (Kaufmanns-UND) wird aber wärmstens empfohlen.

BEISPIEL 2.33: Addition von zwei Zeichenketten

VB
```
Dim s1 As String = "Hallo ", s2 As String = " .NET Freunde!"
MessageBox.Show(s1 & s2)
```

VB 2015 ermöglicht nun auch Stringliterate die sich über mehrere Zeilen erstrecken.

BEISPIEL 2.34: Stringliterate über mehrere Zeilen

VB
```
Dim s = "Viele Grüße,
         Euer Hans"
```

Das sieht doch besser aus als die alte Schreibweise:

```
Dim s = "Viele Grüße," & vbCrLf & "Euer Hans"
```

2.3.4 Date

Datums-/Zeitwerte werden in Variablen vom *Date*-Typ als ganze 64-Bit-Zahlen (8 Bytes) gespeichert. Der Bereich umfasst den 01. Januar des Jahres 1 unserer Zeitrechnung bis zum 31. Dezember 9999 und Uhrzeiten von 0:00:00 (Mitternacht) bis 23:59:59.

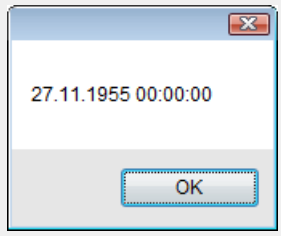
Ein *Date*-Literale muss durch das Rautenzeichen (#) eingeschlossen sein und im Format *m/d/yyyy* (US-amerikanische Schreibweise) angegeben werden.

Werden *Date*-Werte in den *String*-Typ konvertiert, wird das Datum entsprechend des in der Systemsteuerung des Computers eingestellten **kurzen** Datumsformats dargestellt, die Uhrzeit entspricht dem eingestellten Zeitformat (12 oder 24 Stunden).

BEISPIEL 2.35: Der 27.11.1955 wird einer Datumsvariablen zugewiesen und in einer MessageBox angezeigt.

VB
```
Dim dat As Date = #11/27/1955#        ' Reihenfolge: Monat/Tag/Jahr
MessageBox.Show(dat.ToString)
```

Ergebnis



Wollen Sie zusätzlich zum Datum die Uhrzeit mit angeben, muss diese dahinter im Format *hh:mm:ss AM|PM* notiert werden.

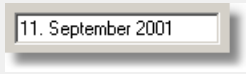
BEISPIEL 2.36: Der 17. August 1987, 19.00 Uhr

VB
```
Dim dat As Date = #8/17/1987 7:00:00 PM#
```

Um auch deutsche Datumsstrings zuweisen zu können, können Sie z.B. die *CDate*-Typkonvertierungsfunktion verwenden.

BEISPIEL 2.37: Ein Datum wird über eine *TextBox* zugewiesen.

VB




```
Dim dat As Date = CDate(TextBox1.Text)
```

HINWEIS: Weitere Informationen über Datums-/Zeitfunktionen bzw. die Methoden der *Date*-Klasse entnehmen Sie dem Kapitel 4.

2.3.5 Boolean

Ein *Boolean* beansprucht satte 16 Bit, obwohl eigentlich nur ein einziges Bit zum Speichern von *True* bzw. *False* reichen würde, aber hinter dieser scheinbar unglaublichen Verschwendung stecken rechentechnische Gründe. Standardmäßig wird eine *Boolean*-Variable bei ihrer Deklaration mit *False* initialisiert.

False entspricht einem Integer-Wert von 0, wenn Sie *Boolean* in *Integer* konvertieren, *True* entspricht dem *Integer*-Wert -1. Etwas anders sieht es aus, wenn Sie umgekehrt einen Integer in einen *Boolean* konvertieren wollen, denn dann wird jeder von null abweichende Wert zu *True* (siehe auch Typkonvertierung im Abschnitt 2.4.8).

2.3.6 Object

Dieser universelle Datentyp spielt als "Mutter für alles" eine Sonderrolle, denn ihm können Sie beliebige Datentypen zuweisen. Eine *Object*-Variable ist ein so genannter *Referenztyp*, denn sie speichert nicht den tatsächlichen Wert, sondern lediglich einen 4 Byte großen Zeiger auf die Adresse der zugewiesenen Variablen.

Vor der Ausführung arithmetischer Operationen und bei eingeschalteter strenger Typprüfung (*Option Strict On*) müssen Sie einen *Object*-Typ mit *CType* (siehe 2.4.8) immer in den gewünschten Typ konvertieren.

BEISPIEL 2.38: Zwei Objektvariablen referenzieren eine *String*- bzw. eine *Integer*-Variable und werden (nach erfolgter Typkonvertierung) zur ersten Objektvariablen addiert. Das Ergebnis wird in einen String konvertiert und angezeigt.

VB

```
Dim s As String = "16"
Dim i As Integer = 20
Dim o1 As Object = s
Dim o2 As Object = i
o1 = CType(o1, Integer) + CType(o2, Integer)     ' referenziert Integer mit Wert 36
MessageBox.Show(o1.ToString)                     ' Ergebnis ist "36"
```

HINWEIS: Den *Object*-Datentyp sollten Sie nur in Ausnahmefällen verwenden, da mit ihm die Typsicherheit (trotz *Option Strict On*) verloren geht (Fehlerquelle!).

2.4 Konvertieren von Datentypen

Visual Basic nimmt es mit den Datentypen sehr genau, zumindest wenn wir "sauber" programmieren wollen und dazu die *Option Strict On* eingeschaltet haben.

2.4.1 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 2.39: Die Zuweisung *Byte* zu *Integer* funktioniert problemlos.

VB
```
Dim b As Byte = 100
Dim i As Integer = b
```

Geradezu oberlehrerhaft verhält sich .NET im umgekehrten Fall. Egal ob der Wert in den kleineren Datentyp passen würde oder nicht – es wird halt gemeckert.

BEISPIEL 2.40: Obwohl der Wert *100* problemlos von einer *Byte*-Variablen aufgenommen werden könnte, erscheint eine Fehlermeldung.

VB
```
Dim i As Integer = 100
Dim b As Byte = i       ' Fehler!
```

Um den meckernden Compiler zu beschwichtigen, ist eine so genannte *explizite Typkonvertierung* erforderlich, für die es unter VB.NET verschiedene Wege gibt.

- Verwendung einer sprachspezifischen Konvertierungsfunktion, wie z.B. *CByte* oder *CDbl* (siehe Tabelle Seite 106).
- Konvertieren mit der *CType*-Funktion
- Konvertierungen über die *Convert*-Klasse
- Einsatz des *TryCast*-Operators
- Verwenden der *ToString*- bzw. *Parse*- Methode (für Konvertierung in/vom *String*-Datentyp)

BEISPIEL 2.41: Drei explizite Typkonvertierungen mit dem gleichen Ergebnis.

VB
```
Dim i As Integer = 100
Dim b As Byte = CByte(i)            ' Variante 1
Dim b As Byte = CType(i, Byte)      ' Variante 2
```

BEISPIEL 2.41: Drei explizite Typkonvertierungen mit dem gleichen Ergebnis.

```
Dim b As Byte = Convert.ToByte(i)    ' Variante 3
```

HINWEIS: Zur Anwendung der Methoden *ToString* und *Parse* siehe 2.4.5, zum *TryCast*-Operator siehe 2.4.9.

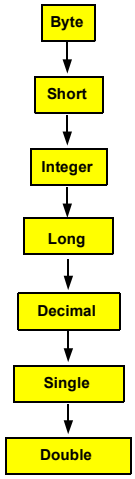
2.4.2 Welcher Datentyp passt zu welchem?

Bei einer impliziten Konvertierung unter *Option Strict On* kann stets nur der "schmalere" der beiden Datentypen in einen "breiteren" umgewandelt werden[1].

BEISPIEL 2.42: Implizite Typkonvertierung *Byte* in *Double*

VB
```
Dim d As Double
Dim b As Byte = 100
d = b                       ' d erhält den Wert 100
```

In der folgenden Abbildung entnehmen Sie der Pfeilrichtung, welcher Typ automatisch welchen Typ aufnehmen kann.



In obiger Abbildung werden Sie die Datentypen *Char*, *String* und *Boolean* vermissen. Für die beiden letzteren ist generell keinerlei automatische Typkonvertierung möglich, hier bleibt Ihnen nur die explizite Konvertierung.

Die implizite (sprich automatische) Konvertierung eines *Char* in einen *String* ist jedoch möglich.

[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 drin sind!

BEISPIEL 2.43: Umwandlung eines Zeichens in einen String

```
VB Dim zeichen As Char = "A"c
   Dim s As String = zeichen          ' s ergibt sich zu "A"
```

2.4.3 Konvertierungsfunktionen

Zur expliziten Typkonvertierung stellt Visual Basic gnädigerweise für jeden fundamentalen Datentyp eine Funktion zur Verfügung (siehe Tabelle).

Konvertierungsfunktion	Datentyp	Konvertierungsfunktion	Datentyp
CShort(*Ausdruck*)	*Short*	CBool(*Ausdruck*)	*Boolean*
CInt(*Ausdruck*)	*Integer*	CByte(*Ausdruck*)	*Byte*
CLng(*Ausdruck*)	*Long*	CChar(*Ausdruck*)	*Char*
CDec(*Ausdruck*)	*Decimal*	CDate(*Ausdruck*)	*Date*
CSng(*Ausdruck*)	*Single*	CStr(*Ausdruck*)	*String*
CDbl(*Ausdruck*)	*Double*	CObj(*Ausdruck*)	*Object*

BEISPIEL 2.44: Alle drei Variablen haben unterschiedliche Datentypen.

```
VB Dim a As Integer = 10000, b As Double = 10000000000000.0, c As Single
   c = CSng(a + b)
   MessageBox.Show(b.ToString)   '  zeigt "1E+13"
```

BEISPIEL 2.45: Ein *True* entspricht hier dem Integer-Wert -1 (*False* ist 0).

```
VB Dim b As Boolean = True
   Dim i As Integer
   i = CInt(b)             ' i erhält den Wert -1
```

HINWEIS: Während bei impliziten Typkonvertierungen nichts verloren geht, können bei expliziten Konvertierungen durchaus Genauigkeitsverluste auftreten.

BEISPIEL 2.46: Bei der Konvertierung *Double* zu *Integer* werden die Nachkommastellen gerundet.

```
VB Dim i As Integer
   Dim d As Double = 12.7456789
   i = CInt(d)             ' i erhält den Wert 13
```

BEISPIEL 2.47: Eine *Integer*-Zahl ungleich null wird immer zu *True* konvertiert.

```
VB Dim b As Boolean
   Dim i As Integer = 10
   b = CBool(i)           ' b erhält den Wert True
```

BEISPIEL 2.48: Wird ein *String* mit *CChar* in eine *Char*-Variable kopiert, so erhält diese das erstemodulglobal Zeichen des Strings.

VB
```
Dim s As String = "Hallo"
Dim z As Char = CChar(s)      ' z erhält den Wert "H"
```

2.4.4 CType-Funktion

Mit Hilfe dieser Funktion können Sie die explizite Konvertierung der verschiedenen Standarddatentypen vornehmen.

SYNTAX: `CType(expr As Object, type As Object)`

expr = zu konvertierender Ausdruck; *type* = Typbezeichner des Zieltyps

BEISPIEL 2.49: Konvertieren eines *Double*- in einen *Integer*-Typ

VB
```
Dim i As Integer
Dim d As Double = 12.7456789
i = CType(d, Integer)            ' i erhält den Wert 13
```

2.4.5 Konvertieren von Strings

Beim *String*-Datentyp scheint es zunächst ähnlich trübe wie bei *Boolean* auszusehen. 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!

BEISPIEL 2.50: Anzeige einer Gleitkommazahl

VB
```
Dim d As Double = 12.75
MessageBox.Show(d.ToString)
```

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 2.51: Einem *Char* wird das zweite Zeichen eines *String* zugewiesen.

```
VB  Dim name As String = "Max"
    Dim c As Char = 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 2.52: Das geht leider nicht[1]

```
VB  Dim s As String = "5"
    Dim i As Integer = s           ' Fehler!
```

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

BEISPIEL 2.53: Das Vorgängerbeispiel kann wie folgt sauber gelöst werden

```
VB  Dim s As String = "5"
    Dim I As Integer = Convert.ToInt32(s)
    MessageBox.Show(i.ToString)      ' zeigt "5"
```

BEISPIEL 2.54: Ein *String* wird in eine *Double*-Zahl konvertiert

```
VB  Dim s As String = "23,50"
    Dim d As Double = Convert.ToDouble(s)    ' d erhält den Wert 23,50
```

Alternativ kann auch die *Parse*-Methode eingesetzt werden (siehe 2.4.7):

BEISPIEL 2.55: Konvertieren eines Stringliterals in eine Ganzzahl.

```
VB  Dim nr As Integer = 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 Übergabewerte als Zeichenketten vorliegen (*Text*-Eigenschaft der Ein- und Ausgabefelder), müssen sie zunächst in Zahlentypen umgewandelt werden, um dann nach ihrer Verarbeitung wieder in Zeichenketten zurückverwandelt und zur Anzeige gebracht zu werden.

[1] Wir beziehen uns auf *Option Strict On*

BEISPIEL 2.56: Ein Ausschnitt aus dem Einführungsbeispiel 1.7.2

VB
```
euro = Convert.ToSingle(TextBox1.Text)          ' Eingabe: String => Single
dollar = euro * kurs                            ' Verarbeitung
TextBox2.Text = dollar.ToString("#,##0.00")     ' Ausgabe: Single => String
```

2.4.6 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`*`(`*`expr`* `As Object)`

typeMethode = eine der Konvertierungsmethoden (*ToBoolean*, *ToByte, ToInt32, ...*)

expr = zu konvertierender Ausdruck

BEISPIEL 2.57: Ein *String* wird in eine *Double*-Zahl konvertiert.

VB
```
Dim s As String = "55,7"
Dim d As Double = Convert.ToDouble(s)
MessageBox.Show(d.ToString)                ' zeigt "55,7"
```

BEISPIEL 2.58: *Boolean* wird in *Integer* und in *String* konvertiert

VB
```
Dim b As Boolean = True
Dim i As Integer = Convert.ToInt32(b)     ' 1
b = False
i = Convert.ToInt32(b)                    ' 0
Dim s As String = Convert.ToString(b)     ' "False"
```

2.4.7 Die Parse-Methode

Die numerischen Typen *Byte*, *Integer*, *Single* und *Double* verfügen u.a. über eigene (statische) *Parse*-Methoden, welche die Stringdarstellung einer Zahl in den entsprechenden Typ konvertieren kann.

BEISPIEL 2.59: Der Inhalt einer *TextBox* wird in eine Gleitkommazahl konvertiert.

VB
```
Dim z As Double = Double.Parse(TextBox1.Text)
```

BEISPIEL 2.60: Konvertieren eines Stringliterals in eine Ganzzahl

VB
```
Dim nr As Integer = Int32.Parse("12")
```

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

2.4.8 Boxing und Unboxing

Die Begriffe *Boxing/Unboxing* gehören zu den häufig strapazierten .NET-Schlagwörtern. 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 2.61: Boxing und Unboxing

VB

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

```
Dim b1 As Boolean = True
Dim o As Object = b1                      ' ok, implizite Konvertierung (Boxing)
Dim b2 As Boolean  = o                    ' Fehler, implizite Konvertierung schlägt fehl
Dim b2 As Boolean = Convert.ToBoolean (o)  ' ok, explizite Konvertierung (Unboxing, True)
```

Um den tieferen Sinn von Boxing/Unboxing zu verstehen, sollten Sie sich in Abschnitt 2.2.2 nochmals den Unterschied zwischen den beiden fundamentalen Arten von Datentypen verdeutlichen, d.h., zwischen Werte- und den Verweistypen.

Boxing

Es erhebt sich die Frage, was denn passiert, wenn man einer *Object*-Variablen einen Wertetyp zuweist, der naturgemäß im Stack gespeichert ist.

BEISPIEL 2.62: Ein Integer wird einem *Object*-Datentyp zugewiesen.

VB

```
Dim i As Integer = 25
Dim o As Object = i
```

Die genaue 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 2.63: Das Vorgängerbeispiel wird fortgesetzt

```
VB
...
Dim j As Integer = o          ' Fehler!
Dim j As Integer = CInt(o)    ' ok
```

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 2.64: Ja, auch das funktioniert!

```
VB
Dim i As Integer = New Integer()
i = 12
```

2.4.9 TryCast-Operator

Eine weitere Alternative zur expliziten Typumwandlung (bzw. Unboxing) bietet der *TryCast*-Operator, der allerdings nur auf Verweis- und nicht auf Wertetypen anwendbar ist. Auch alle Steuerelemente gehören zu den Verweistypen.

BEISPIEL 2.65: Konvertieren des *sender*-Parameters eines Eventhandlers.

```
VB
Me.Text = (TryCast(sender, TextBox)).Text
```

Misslingt die Konvertierung, so wird kein Fehler ausgelöst, sondern der Variablen wird der Wert *Nothing* zugewiesen.

2.4.10 Nullable Types

Ein weiterer Unterschied zwischen Wertetypen, wie *Integer* oder *Structure*, und Referenztypen, wie *Form* oder *String*, ist der, dass Referenztypen so genannte Null-Werte unterstützen. Eine Referenztyp-Variable kann also den Wert *Nothing* enthalten, d h., die Variable referenziert im Moment keinen bestimmten Wert. Demgegenüber enthält eine Wertetyp-Variable immer einen Wert, auch wenn dieser, wie bei einer *Integer*-Variablen, den Wert 0 (null) hat. Falls Sie einer Wertetyp-Variablen *Nothing* zuweisen, wird diese auf ihren Default-Wert zurückgesetzt, bei einem *Integer* wäre das 0 (null)).

Der Compiler kann aber durch ein der Typdeklaration nachgestelltes Fragezeichen (?) einen Wertetyp in eine generische *System.Nullable(Of T As Structure)*-Struktur verpacken.

BEISPIEL 2.66: Einige Deklarationen von Nullable-Typen

```
VB
Dim i As Integer? = 10
Dim j As Integer? = Nothing
Dim k As Integer? = i + j             ' Nothing
```

BEISPIEL 2.67: Ein an die Subroutine *PrintValue* übergebener *Integer*-Wert wird nur angezeigt, wenn ihm ein Wert zugewiesen wurde. Ansonsten erfolgt die Ausgabe "Null Wert".

VB
```
Sub PrintValue(i As Nullable(Of Integer))
    If i.HasValue Then
      Console.WriteLine(CInt(i))
    Else
      Console.WriteLine("Null Wert!")
    End If
End Sub
```

Von Nutzen dürften Nullable-Typen besonders dann sein, wenn Daten aus einer relationalen Datenbank gelesen bzw. dorthin zurückgeschrieben werden sollen (siehe dazu Kapitel 10 und 11).

2.5 Operatoren

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

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

Doch bevor es richtig losgeht sollten wir uns mit dem Begriff *Ausdruck* etwas anfreunden. Man versteht darunter die kleinste ausführbare Einheit eines Programms. Ein Ausdruck setzt zumindest einen Operator voraus und benötigt meist zwei Operanden.

BEISPIEL 2.68: Operatoren

VB

Im Ausdruck

```
i = 12
```

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

2.5.1 Arithmetische Operatoren

Neben den vier Grundrechenarten werden folgende Operatoren unterstützt:

Operator	Beispielausdruck	Erklärung
+	x + y	Addition
-	x - y -x	a) Subtraktion b) negative Zahl

Operator	Beispielausdruck	Erklärung
*	x * y	Multiplikation
/	x / y	Division
\	x \ y	Integer-Division (liefert nur ganzzahligen Anteil)
Mod	x Mod y	Modulo-Division (liefert Restwert)
^	x ^ y	Potenzoperator
&	"a" & "b"	Addition von Zeichenketten

BEISPIEL 2.69: Die Ergebnisse einiger arithmetischer Operationen werden kommentiert.

VB
```
Dim i As Integer, d As Double
i = 3 *(4 + 5) * 6       ' 162
d = 3 ^ 2                ' 9
d = 7 / 3                ' 2.3333333333333
d = 7 \ 3                ' 2 (ganzzahlige Division!)
i = 7 Mod 3              ' 1 (Rest!)
```

2.5.2 Zuweisungsoperatoren

In Ergänzung zum normalen Zuweisungsoperator (=) gibt es für fast alle arithmetischen Operatoren eine Kurzformnotation (siehe folgende Tabelle).

Operator	Beispielausdruck	Erklärung
=	x = y	x wird der Wert von y zugewiesen
+=	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 zu x / y (Nachkommastellen bleiben)
\=	x \= y	x ergibt sich zu x \ y (Nachkommastellen abgeschnitten)
^=	x ^= y	x wird mit y potenziert
&=	x &= y	an String x wird String y angehängt

Der einfache Zuweisungsoperator ist für den Einsteiger immer etwas problematisch, da er sich rein äußerlich nicht vom Vergleichsoperator unterscheidet.

BEISPIEL 2.70: Das erste "=" wird als Zuweisungs-, das zweite "=" hingegen als Vergleichsoperator interpretiert, d.h., *x* wird *True*, falls *y* gleich *z* ist, sonst *False*.

VB
```
x = y = z
```

Um die Lesbarkeit zu verbessern, sollte man den Vergleichsausdruck klammern:

```
x = (y = z)
```

Die verkürzten Zuweisungsoperatoren nur bringen relativ bescheidene Verbesserungen.

BEISPIEL 2.71: Verkürzter Zuweisungsoperator

VB

Anstatt

```
i = i + 1
```

kann man schreiben

```
i += 1
```

2.5.3 Logische Operatoren

Logische Operatoren verknüpfen zwei Boolesche Variablen bzw. Ausdrücke miteinander, um daraus einen neuen *True-/False*-Wert zu ermitteln.

Operator	Beispielausdruck	Erklärung
And	*x And y*	**Und**: liefert *True*, wenn beide Operanden *True* sind
Or	*x Or y*	**Oder**: liefert *True*, wenn mindestens einer der Operanden *True* ist
Xor	*x Xor y*	**Exklusiv Oder**: liefert *True*, wenn genau einer der beiden Operanden *True* ist
AndAlso	*x AndAlso y*	**Spezielles Und**: ist der erste Operator *False*, wird der zweite nicht mehr überprüft
OrElse	*x OrElse y*	**Spezielles Oder**: ist der erste Operator *True*, wird der zweite nicht mehr überprüft
Not	*Not x*	**Negation**: negiert den Wahrheitswert

Hier die Wahrheitstabelle der wichtigsten zweistelligen logischen Operationen:

Operand 1	Operand 2	And	Or	Xor
False	*False*	*False*	*False*	*False*
False	*True*	*False*	*True*	*True*
True	*False*	*False*	*True*	*True*
True	*True*	*True*	*True*	*False*

BEISPIEL 2.72: Der Variablen *b* wird der Wert *True* zugewiesen[1].

VB

```
Dim b As Boolean = True And True Or False And False
```

[1] Umständlicher geht es sicherlich kaum, aber Sie wollen ja schließlich etwas lernen!

2.5.4 Vergleichsoperatoren

Vergleichs- oder relationale Operatoren vergleichen zwei Ausdrücke miteinander und liefern als Ergebnis einen Wahrheitswert. In Visual Basic ist das übliche Angebot enthalten.

Operator	Beispielausdruck	Erklärung
=	x = y	Arithmetische Vergleiche. (der Stringvergleich wird von links nach rechts entsprechend dem ANSI-Code der Zeichen durchgeführt)
<	x < y	
<=	x <= y	
>	x > y	
>=	x >= y	
<>	x <> y	

BEISPIEL 2.73: Und-Verknüpfung von zwei Ausdrücken.

VB
```
Dim x As Integer = 5, b As Boolean
b = (x < 5) And (x >= 0)                ' b ist False
```

Im obigen Beispiel steht das Ergebnis bereits nach dem Auswerten des ersten Ausdrucks fest. Man könnte also Rechenzeit einsparen, wenn man auf das Auswerten des zweiten Ausdrucks verzichten würde. Um ein solches Verhalten zu erreichen, stehen die logischen Operatoren *AndAlso* und *OrElse* zur Verfügung.

BEISPIEL 2.74: Das Vorgängerbeispiel wird rationeller programmiert.

VB
```
Dim x As Integer = 5, b As Boolean
b = (x < 5) AndAlso (x >= 0)            ' b ist False
```

2.5.5 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
()
^
–
/
\
Mod

Operator
+ -
= < >
<> <= >=
Not
And AndAlso *Or OrElse*
Xor

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

BEISPIEL 2.75: Arithmetische Operationen

```
VB  Dim y As Double = 2 ^ 2 + 1 + 2 / 4              ' y = 5,5
```

BEISPIEL 2.76: Boolesche Operationen

```
VB  Dim b As Boolean = Not True And False Or 5 > 6    ' b = False
```

HINWEIS: Durch Klammern kann die hierarchische Reihenfolge außer Kraft gesetzt werden[1].

2.6 Kontrollstrukturen

Schleifen- und Verzweigungsanweisungen unterbrechen den linearen Programmablauf und gehören zum Einmaleins des Programmierens.

2.6.1 Verzweigungsbefehle

"Programmweichen" werden durch Entscheidungsanweisungen (Verzweigungen) gestellt. Die folgende Tabelle gibt einen Überblick über die zum "Basic-Urgestein" zählenden Entscheidungsbefehle.

Verzweigungsanweisung	Erklärung

`If` *Bedingung* `Then` *Anweisungen*
`[Else` *Anweisungen*`]`
`End If`
	Bedingte Verzweigung, wenn die gesamte *If Then*-Anweisung in einer einzigen Zeile steht, kann auf *End If* verzichtet werden.

`If` *Bedingung1* `Then`
Anweisungen
`[ElseIf` *Bedingung2* `Then`
	Blockstruktur *If...ElseIf...End If*
	Jede Zeile muss mit *Then* enden!

[1] Da Sie aber in der Schule gut aufgepasst haben, werden wir auf weitere Beispiele verzichten.

Verzweigungsanweisung	Erklärung

Anweisungen
`ElseIf` *Bedingung3* `Then`
Anweisungen ...]
[`Else` *Anweisungen*]
`End If`
	Else-Anweisungen werden dann ausgeführt, wenn keine der *If*- bzw. *ElseIf*-Bedingungen zutrifft.

`Select Case` *Ausdruck* `Case` *Ausdruck1* *Anweisungen* [`Case` *Ausdruck2* *Anweisungen ...*] [`Case Else` *Anweisungen*] `End Select`	Blockstruktur *Select Case/Case/End Select* Der Ausdruck kann eine Variable oder ein beliebiger Ausdruck sein, der mit den hinter *Case* angeführten Ausdrücken verglichen wird. Nach erstem Erfolg wird der Block verlassen!

In den meisten Fällen werden Sie zum Prüfen von Bedingungen die *If-Then*-Anweisung verwenden.

HINWEIS: Wenn Sie die *If-Then*-Anweisung in **einer** Programmzeile unterbringen, braucht das Blockende nicht mit *End If* markiert zu werden.

BEISPIEL 2.77: In dieser einzeiligen *If-Then*-Anweisung wird im *Label* "Verbessern!" angezeigt.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then Label1.Text = "Gratuliere!" Else Label1.Text = "Verbessern!"
```

BEISPIEL 2.78: Die Alternative für das Vorgängerbeispiel braucht zwar mehr Platz, ist aber übersichtlicher.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then
   Label1.Text = "Gratuliere!"
Else
   Label1.Text = "Verbessern!"            ' zutreffende Bedingung
End If
```

Optional können Sie im *If-Then*-Block noch *ElseIf*- und *Else*-Zweige verwenden, wobei die *ElseIf*-Bedingung nur dann geprüft wird, wenn keine der vorstehenden *If*-Bedingungen erfüllt war.

BEISPIEL 2.79: Im *Label* wird "Befriedigend" angezeigt.

VB
```
Dim zensur As Byte = 3
If zensur = 1 Then
   Label1.Text =  "Sehr gut!"
ElseIf zensur = 2 Then
   Label1.Text =  "Gut"
```

BEISPIEL 2.79: Im *Label* wird "Befriedigend" angezeigt.

```
ElseIf zensur = 3 Then                ' zutreffende Bedingung
   Label1.Text =  "Befriedigend"
 '(usw.)
End If
```

Mit *Select Case* wird ein Ausdruck auf mehrere mögliche Ergebnisse hin überprüft. Im Testausdruck kann ein beliebiger arithmetischer oder logischer Ausdruck stehen.

BEISPIEL 2.80: Diese Kontrollstruktur leistet das Gleiche wie das Vorgängerbeispiel.

VB
```
Dim zensur As Byte = 3
Select Case zensur
   Case 1: Label1.Text = "Sehr gut"
   Case 2: Label1.Text =  "Gut"
   Case 3: Label1.Text =  "Befriedigend"     ' zutreffende Bedingung
   '(usw.)
End Select
```

In einem *Case*-Zweig können auch mehrere Bedingungen geprüft werden.

BEISPIEL 2.81: Das *Label* zeigt "Frühling" an.

VB
```
Dim monat As Byte = 5
Select Case monat
   Case 12,1,2: Label1.Text =  "Winter"
   Case 3,4,5: Label1.Text =  "Frühling"       ' zutreffende Bedingung
   Case 6,7,8: Label1.Text =  "Sommer"
   Case 9,10,11: Label1.Text =  "Herbst"
   Case Else
      Label1.Text =  "kein gültiger Monat!"
End Select
```

Alternativ zum Aufzählen mehrerer Bedingungen kann mittels *To* auch ein Bereich angegeben werden.

BEISPIEL 2.82: Ein Zweig aus dem Vorgängerbeispiel könnte wie folgt ersetzt werden.

VB
```
...
  Case 3 To 5: Label1.Text = "Frühling"
...
```

HINWEIS: Sie sollten, wo immer es geht, anstelle einer *If-Then*-Anweisung mit eingeschachtelten *ElseIf*-Verzweigungen eine *Select Case*-Anweisung verwenden. *Select Case* wird wesentlich schneller ausgeführt, da die Prüfbedingung nur einmal auszuwerten ist.

2.6.2 Schleifenanweisungen

Visual Basic kennt zwei Grundtypen[1]:

- *For-Next*- und
- *Do-Loop*-Schleifen .

Die folgende Tabelle gibt einen Überblick über alle möglichen Schleifenkonstruktionen:

Schleifenanweisung	Erklärung

`For` *Zähler=Anfangs* `To` *Endwert* `[Step` *Schritt*`]`
  Anweisungen
  `[Exit For]`
  Anweisungen
`Next [`*Zähler*`]`
	For...Next-Zählschleife,
	Abbruch mit *Exit For*, ohne *Step* ist Schritt 1

`Do [While|Until` *Bedingung*`]`
  Anweisungen
  `[Exit Do]`
  Anweisungen
`Loop`
	Do While...Loop-Bedingungsschleife,
	Abbruchbedingung am Schleifenanfang

`Do`
  Anweisungen
  `[Exit Do]`
  Anweisungen
`Loop [While|Until` *Bedingung*`]`
	Do...Loop While-Bedingungsschleife,
	Abbruchbedingung am Schleifenende

`While` *Bedingung* *Anweisungen* `End While`	gleichbedeutend mit *Do While ... Loop*-Bedingungsschleife

For-Next-Schleifen

Bei diesem klassischen Schleifentyp wird die Zählervariable automatisch hochgezählt (inkrementiert) bzw. heruntergezählt (dekrementiert).

BEISPIEL 2.83: Zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* ausgeben

VB

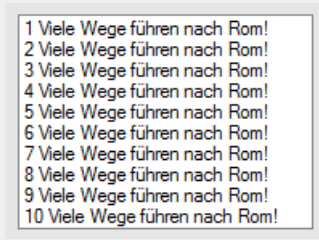
```
Dim i As Integer
For i = 1 To 10
       ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
Next i
```

Die Angabe der Zählervariablen *i* nach *Next* kann auch weggelassen werden, wird aber aus Gründen der Übersichtlichkeit empfohlen.

[1] Auf *For-Each*-Schleifen gehen wir en passant erst im Rahmen der objektorientierten Programmierung ein, siehe auch Praxisbeispiel 2.9.3.

BEISPIEL 2.83: Zehnmal untereinander den laufenden Index und einen Text in einer *ListBox* ausgeben

Ergebnis



Do-Loop-Schleifen

Schleifen dieses Typs erlauben eine flexible Programmierung, da die Abbruchbedingung sehr variabel ist. Um das Inkrementieren/Dekrementieren der Zählervariablen muss man sich allerdings selbst kümmern.

Bei *While* wird die Schleife ausgeführt, **während** die Bedingung erfüllt ist, bei *Until* nur so lange, **bis** die Bedingung erfüllt ist.

In Abhängigkeit davon, ob die Abbruchbedingung am Schleifenanfang oder an deren Ende kontrolliert wird, spricht man auch von *kopfgesteuerten* bzw. *fußgesteuerten* Schleifen.

BEISPIEL 2.84: Ein identisches Resultat wie die obige *For-Next*-Schleife ergibt die folgende *Do While ... Loop*-Schleife

VB

```
Dim i As Integer = 1
Do While i <= 10
   ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
   i += 1
Loop
```

Ein umfassendes Testprogramm mit weiteren möglichen *Do ... Loop*-Schleifenkonstruktionen finden Sie im Praxisbeispiel

► 2.9.3 Schleifenanweisungen kennen lernen

2.7 Benutzerdefinierte Datentypen

Sie sind als Programmierer natürlich nicht nur auf die einfachen Datentypen *Integer, Single, ...* angewiesen, sondern können auch selbst neue, komplexere Datentypen kreieren. Wir wollen in diesem Abschnitt Aufzählungstypen (Enumerationen) und strukturierte Datentypen behandeln.

2.7.1 Enumerationen

Sammlungen von miteinander verwandten Konstanten können in so genannten Enumerationen (*Enum*s) zusammengefasst werden.

HINWEIS: Alle in der Enumeration enthaltenen Konstanten müssen vom gleichen Datentyp sein, zulässig sind nur *Byte*, *Short*, *Integer* und *Long*.

SYNTAX:
```
[Private|Public] Enum Name As Datentyp
        Konstante1 = wert1
        Konstante2 = wert2
        ...
    End Enum
```

Falls die Angabe des Datentyps weggelassen wird, handelt es sich automatisch um *Integer*-Konstanten.

BEISPIEL 2.85: Eine Enumeration für drei Konstanten vom *Byte*-Datentyp

VB
```
Public Enum erstesQuartal As Byte
       JANUAR = 1
       FEBRUAR
       MÄRZ
End Enum
```

Es genügt, wenn nur der ersten Mitgliedskonstanten (*JANUAR*) ein (*Byte*-)Wert zugewiesen wird, der Wert der Nachfolger wird automatisch um eins erhöht.

Auf die deklarierten Konstanten kann direkt zugegriffen werden.

BEISPIEL 2.86: Verwendung der oben deklarierten Enumeration

VB
```
Const monat As Byte = erstesQuartal.FEBRUAR
```

Auch eine vorherige Variablendeklaration ist möglich.

BEISPIEL 2.87: Eine Modifikation des Vorgängerbeispiels

VB
```
Dim quartal As erstesQuartal
Const monat As Byte = quartal.FEBRUAR
```

2.7.2 Strukturen

Mit der *Structure*-Anweisung definieren Sie komplexe Datentypen, die ein oder auch mehrere Element(e) enthalten dürfen.

HINWEIS: In diesem Abschnitt wollen wir Strukturen nicht tiefgründiger behandeln, sondern nur einen ersten Einblick gewähren.

Deklaration

Vom Prinzip her entspricht die Definition einer *Struktur* der einer *Klasse* (siehe Kapitel 3), allerdings sind Strukturen *Wertetypen*, während Klassen zu den *Verweis- bzw. Referenztypen* gehören.

Eine einfache Struktur hat folgenden Aufbau:

```
SYNTAX: [Private|Public] Structure Name
            Dim  Mitglied1 As Datentyp
            ...
        End Structure
```

Beachten Sie:

- Die *Structure*-Anweisung ist auf lokaler bzw. Prozedurebene unzulässig und nur auf globaler Ebene anwendbar.
- Damit die Elemente einer Struktur von außerhalb sichtbar sind, muss der *Public*-Modifizierer vorangestellt werden.[1]

BEISPIEL 2.88: Structure-Deklaration

VB

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

```
Public Structure Person
    Dim Vorname, Nachname As String
    Dim Alter As Byte
End Structure
```

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 2.89: *(Fortsetzung)* Sie erzeugen zwei Variablen des oben definierten Datentyps *Person*:

VB

```
Dim person1, person2 As Person
```

Datenzugriff

Um auf den Wert einer Strukturvariablen zuzugreifen, müssen Name und Element durch einen Punkt (so genannter *Qualifizierer*) voneinander getrennt sein[2].

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

[2] Dies entspricht der Schreibweise beim Zugriff auf Objekteigenschaften.

BEISPIEL 2.90: Fortsetzung des Vorgängerbeispiels

VB
```
Dim a As Integer, name As String
person1.Vorname = "Max"        ' Schreibzugriff
person1.Nachname = "Müller"
person1.Alter = 50
a = person1.Alter              ' Lesezugriff
name = person1.Vorname & "  " & person1.Nachname
```

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

BEISPIEL 2.91: Die Variable *person1* wird "geklont".

VB
```
person2 = person1
```

With-Anweisung

Ein vereinfachter Zugriff auf Strukturvariablen ist mit der *With- End With*-Anweisung möglich.

SYNTAX:
```
With Strukturvariable
  ' Anweisungen
End With
```

BEISPIEL 2.92: Das Vorgängerbeispiel könnte auch so geschrieben werden:

VB
```
Dim a As Integer, name As String
With person1
   .Vorname = "Max"
   .Nachname = "Müller"
   .alter = 50
   a = .alter
   name = .Vorname & "  " & .Nachname
End With
```

HINWEIS: Mit der *With*-Anweisung kann auch der Zugriff auf Eigenschaften und Methoden beliebiger Objekte vereinfacht bzw. übersichtlicher gestaltet werden (siehe dazu Kapitel 3).

Bemerkungen

Wie bereits erwähnt, 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*, *Nothing* oder *False* initialisiert.

BEISPIEL 2.93: Die Strukturvariable *person1* wird mit ihrem Standard-Konstruktor erzeugt und initialisiert. Das Feld *alter* hat automatisch den Wert *0*.

VB
```
Dim person1 As New Person()
Dim a As Integer = person1.Alter              ' a erhält 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*), der Heap ist für Wertetypen tabu!

2.8 Nutzerdefinierte Funktionen/Prozeduren

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

Funktionen und Prozeduren sind unter Visual Basic zunächst das, was man in den Anfangszeiten der Programmierung als "Unterprogramm" bezeichnet hat. Später, im OOP-Kapitel 3, werden Sie feststellen, dass Funktionen und Prozeduren zu Methoden einer Klasse mutiert sind.

2.8.1 Deklaration und Syntax

Falls Sie nur an einem einzigen Rückgabeparameter interessiert sind, sollten Sie einer Funktion den Vorzug vor einer Prozedur geben, andernfalls ist meistens eine Prozedur die richtige Entscheidung.

Function

Funktionen deklarieren Sie mit dem Schlüsselwort *Function*. Als Funktionstyp bzw. Rückgabewert kommt eigentlich jeder Datentyp in Frage. Jede *Function* hat einen Namen und einen Körper. Letzterer enthält die beim Methodenaufruf auszuführenden Anweisungen. Ein vorzeitiges Verlassen ist mit *Exit Function* möglich. Den Rückgabewert können Sie entweder mittels *Return* übergeben oder aber direkt in den Funktionsnamen schreiben.

SYNTAX:
```
[Private|Public] Function Name ([Parameterliste]) As Datentyp
    ' Code definieren
    ' ...
    [Exit Function]       ' Vorzeitiges Verlassen
    ' ...
    Return Ausdruck       ' Rückgabe des Funktionswertes
End Function
```

Die *Parameterliste* ist eine durch Kommas separierte Liste mit den einzelnen Parametern (Argumenten), von denen jeder wie folgt notiert wird:

SYNTAX:
```
[ByVal|ByRef] Parameter [As Datentyp]
```

Zur Bedeutung der Übergabeart *ByVal* bzw. *ByRef* siehe Abschnitt 2.8.3.

BEISPIEL 2.94: Funktion zur Berechnung des Kugelgewichts. Übergabeparameter sind der Radius und das spezifische Gewicht.

VB
```
Private Function Kugel(ra As Double, sg As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Beim Aufruf der Funktion müssen Reihenfolge und Datentyp der Parameter exakt übereinstimmen. Im aufrufenden Code muss der Rückgabewert einer Variablen gleichen Datentyps wie die Funktion zugewiesen werden.

BEISPIEL 2.95: *(Fortsetzung)* Obige Funktion wird mit einer eisernen Kugel von 20 cm Durchmesser getestet.

VB
```
Dim g As Double = Kugel(10, 7.87)             ' Radius ist 10cm, spez. Gewicht = 7,87 gr/cm3
MessageBox.Show(g.ToString("#,#0.000 Gramm"))  ' das Gewicht der Kugel ist 32.965,779 Gramm
```

Sub

Prozeduren werden mit dem *Sub*-Schlüsselwort (kommt von *Sub*routine) deklariert und haben keinen Rückgabewert. Ansonsten gelten die gleichen syntaktischen Regeln wie bei Funktionen.

SYNTAX:
```
[Private|Public] Sub Name ([Parameterliste])
    ' Code definieren
    ' ...
    [Exit Sub]       ' Vorzeitiges Verlassen
    ' ...
End Sub
```

Da eine Prozedur keinen Rückgabewert hat, arbeitet sie oft mit globalen Variablen zusammen.

BEISPIEL 2.96: Anwenden einer Prozedur zum Berechnen des Kugelgewichts

VB
```
Private gew As Double             ' globale Variable

Private Sub Kugel(ra As Double, sg As Double)
  Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
```

Die globale Variable *gew* wird mit dem Ergebnis gefüttert:

```
  gew = sg * vol
End Sub
```

Beim Aufruf einer Prozedur können Sie, müssen aber nicht, das Schlüsselwort *Call* voranstellen. Wir bevorzugen diese Variante nur gelegentlich in der Lernphase, da so keinerlei Verwechslungsgefahr mit dem Aufruf einer Funktion besteht.

BEISPIEL 2.97: *(Fortsetzung)* Unsere Prozedur wird nun ebenfalls mit der legendären Eisenkugel getestet.

VB
```
Call Kugel(10, 7.87)
MessageBox.Show(gew.ToString("#,#0.000 Gramm"))   '  zeigt "32.965,779 Gramm"
```

2.8.2 Parameterübergabe allgemein

In unseren bisherigen *Kugel*-Beispielen hatten wir beim Aufruf der Funktion bzw. der Prozedur feste Zahlenwerte (Literale) übergeben. In der Praxis treten an diese Stelle aber meistens Variablen oder sogar Ausdrücke.

Signatur der Parameterliste

Normalerweise muss beim Aufruf einer Funktion/Prozedur die so genannte *Signatur* der Parameterliste beachtet werden, d h., die Reihenfolge der Parameter und ihr Datentyp müssen zur deklarierten Parameterliste passen (Ausnahme: benannte Parameter, siehe unten).

BEISPIEL 2.98: *(Fortsetzung)* Die in 2.8.1 deklarierte *Kugel*-Funktion wird aufgerufen.

VB
```
Dim radius As Double = 5
Dim spezGew As Double = 7.87
Dim gew As Double = Kugel(2 * radius, spezGew)
```

Benannte Parameter

Die Verwendung *benannter Parameter* macht es möglich, dass die in der Signatur festgelegte Reihenfolge bei der Parameterübergabe nicht unbedingt eingehalten werden muss. Allerdings sind dann der deklarierte Name des Parameters und die Zeichenfolge ":=" voranzustellen.

BEISPIEL 2.99: Das Vorgängerbeispiel wird mit benannten Parametern ausgeführt, die Reihenfolge der übergebenen Werte ist bewusst vertauscht.

VB
```
Dim radius As Double = 5
Dim spezGew As Double = 7.87
Dim gew As Double = Kugel(sg := spezGew, ra := 2 * radius)
```

HINWEIS: Die Verwendung benannter Parameter vereinfacht die Programmierung besonders dann, wenn Sie Funktionen/Prozeduren aufrufen, die über viele optionale Parameter verfügen (siehe Abschnitt 2.8.4).

2.8.3 Übergabe mit ByVal und ByRef

Jedem Mitglied der Parameterliste einer Funktion/Prozedur ist entweder *ByVal* oder *ByRef* vorangestellt[1]. Dadurch wird festgelegt, ob die Parameterübergabe von Wertetypen "als Wert" (*ByVal*) oder "als Referenz" (*ByRef*) erfolgen soll. Was bedeuten diese Übergabearten?

ByVal

Mit der standardmäßigen *ByVal*-Übergabe wird der Wert des übergebenen Parameters (ein Wertetyp) in eine vom Compiler erzeugte lokale Variable des Funktions- bzw. Prozedurkörpers kopiert. Die aufrufende Variable und die lokale Variable im Inneren der Funktion/Prozedur haben danach nichts mehr miteinander zu tun, eine Veränderung des Wertes der lokalen Variablen bleibt ohne Rückwirkung auf den Übergabeparameter, so dass ein "Zurückgeben" von Ergebnissen – wie im obigen Beispiel gezeigt – nicht möglich ist.

Wenn Sie hingegen *ByRef* übergeben, so verfügt die aufgerufene Funktion/Prozedur nur scheinbar über eine eigene lokale Variable. In Wirklichkeit steht der ursprüngliche Parameter weiter zur Verfügung, eine "Entkopplung" hat nicht stattgefunden.

HINWEIS: Im Interesse der Fehlersicherheit Ihrer Programme sollten Sie Parameter grundsätzlich *ByVal* übergeben (das gilt auch für Referenztypen wie Arrays, Strings und sonstige Objekte). *ByRef* sollten Sie nur dann verwenden, wenn Sie es tatsächlich benötigen und Werte nach außen hin verändern wollen (siehe Praxisbeispiel 2.9.4).

ByRef

Wenn Sie eine Variable *ByRef* übergeben, so wird nicht der Wert, sondern der Zeiger auf die Speicherplatzadresse der Variablen übermittelt. Der aufgerufenen Funktion/Prozedur ist es dadurch möglich, am Wert der Variablen "herumzudoktern". Auf diese Weise können auch Prozeduren ihre Ergebnisse ohne Verwendung globaler Variablen (siehe obiges Beispiel) zurückliefern.

BEISPIEL 2.100: ***By Reference***

VB

In obiger Prozedur zur Kugelgewichtsbestimmung wird die Übergabeart des ersten Parameters in *ByRef* geändert. Dadurch wird es möglich, über diesen Parameter auch das Berechnungsergebnis zu übertragen.

```
Private Sub Kugel(ByRef ra As Double, sg As Double)
  Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
  ra = sg * vol                                    ' Radius wird zu Gewicht!
End Sub
```

Zum Testen der Prozedur: Wir haben den ersten Parameter bewusst nicht mit *radius*, sondern mit *x* bezeichnet, da er seine Bedeutung wechselt und nach dem Aufruf der *Kugel*-Prozedur eigentlich *gewicht* heißen müsste.

[1] *ByVal* kann weggelassen werden, da es Standard ist.

BEISPIEL 2.100: *By Reference*

```
Dim x As Double = 10
Dim spezGew As Double = 7.87

Kugel(x, spezGew)
MessageBox.Show(x.ToString("#,#0.000 Gramm"))     ' zeigt "32.965,779 Gramm"
```

2.8.4 Optionale Parameter

Wenn Sie Parameter mit dem Schlüsselwort *Optional* deklarieren, können Sie diese beim Aufruf auch weglassen, sie werden dann durch Standardwerte aufgefüllt.

BEISPIEL 2.101: Unsere *Kugel*-Funktion wird modifiziert, so dass der zweite Parameter optional ist und den Standardwert 7.87 (spezifisches Gewicht von Eisen) erhält.

VB
```
Public Function Kugel(ra As Double, Optional sg As Double = 7.87) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Für den Aufruf der Funktion gibt es nun zwei Versionen:

```
Dim g As Double = Kugel(10)                       ' Berechnung mit Standardwert (Eisen)
MessageBox.Show(g.ToString("#,#0.000 Gramm"))     ' zeigt "32.965,779 Gramm"
```

oder

```
Dim g As Double = Kugel(10, 11.34)                ' Berechnung für 11.34gr/cm3 (Blei)
MessageBox.Show(g.ToString("#,#0.000 Gramm"))     ' zeigt "47.500,881 Gramm"
```

HINWEIS: Einem optionalen Parameter müssen Sie **immer** einen Standardwert zuweisen!

Eine modernere Alternative zu optionalen Parametern ist das Überladen von Funktionen/Prozeduren (siehe folgender Abschnitt).

2.8.5 Überladene Funktionen/Prozeduren

Wenn mehrere gleichnamige Funktionen/Prozeduren ohne Namenskonflikt friedlich nebeneinander existieren, so spricht man von *überladenen* Funktionen. Die Unterscheidung zwischen ihnen wird vom Compiler anhand der Parameterliste, sprich Signatur, getroffen. Unter Signatur verstehen wir die Reihenfolge der übergebenen Parameter und ihr Datentyp.

HINWEIS: Überladene Funktionen/Prozeduren müssen eine unterschiedliche Signatur haben!

BEISPIEL 2.102: Die *Kugel*-Funktion steht in zwei überladenen Versionen zur Verfügung. Die erste Version verlangt als Parameter den Radius und das spezifische Gewicht.

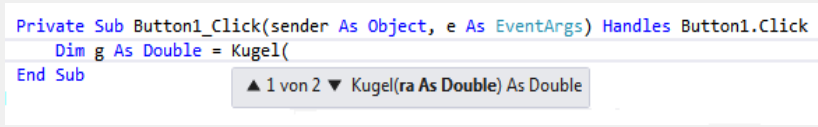
VB

```
Private Function Kugel(ra As Double, sg As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return sg * vol
End Function
```

Eine zweite Version soll nur das Volumen der Kugel berechnen.

```
Private Function Kugel(ra As Double) As Double
    Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
    Return vol
End Function
```

Wenn Sie jetzt in der IDE von Visual Studio die Funktion verwenden wollen, werden Ihnen automatisch beide überladenen Versionen angeboten:



Der Aufruf ist entweder so

```
Dim g As Double = Kugel(10, 7.87)       ' liefert  "32.965,779"
```

oder so möglich

```
Dim v As Double = Kugel(10)             ' liefert  "4.188,790"
```

Wie Sie bestimmt schon bei der Arbeit im Codefenster von Visual Studio bemerkt haben, werden für die meisten Objektmethoden mehr oder weniger viele Überladungen angeboten.

Die *Show*-Methode der *MessageBox* hat beispielsweise insgesamt 21 Überladungen!

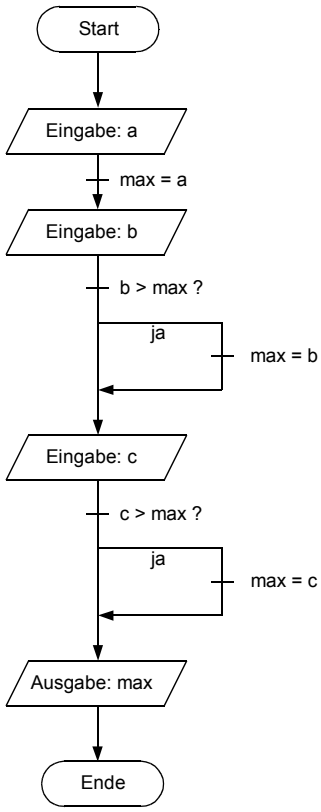
2.9 Praxisbeispiele

2.9.1 Vom PAP zum Konsolen-Programm

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 im Folgenden abgebildete Programmablaufplan (PAP) zeigt die Berechnungsvorschrift (Algorithmus).



Sie erkennen hier die typische EVA-Grundstruktur, bei der die Anweisungen in der Reihenfolge Eingabe, Verarbeitung, Ausgabe ausgeführt werden.

Programmierung

Beim Eintippen der Befehle (z.B. mit Notepad) gibt obiger PAP eine nützliche Orientierung:

```
Imports System
Module Maximum3
  Sub Main()
      Console.WriteLine("Maximum von drei Zahlen")  ' Überschrift
      Console.WriteLine()                           ' Leerzeile
      Dim a, b, c, max As Integer                   ' Variablendeklaration
      Console.WriteLine("Geben Sie die erste Zahl ein!")
      a = CInt(Console.ReadLine())                 ' Eingabe a
      max = a                                      '  Initialisieren von max
      Console.WriteLine("Geben Sie die zweite Zahl ein!")
      b = CInt(Console.ReadLine())                 ' Eingabe b
      If b > max Then max = b                      ' Bedingung
      Console.WriteLine("Geben Sie die dritte Zahl ein!")
      c = CInt(Console.ReadLine())                 ' Eingabe c
```

```
      If c > max Then max = c                    ' Bedingung
      Console.WriteLine("Das Maximum ist " + max.ToString)   ' Ergebnisausgabe
      Console.ReadLine()                        ' Programm wartet auf <Enter>, um zu beenden
   End Sub
End Module
```

Speichern Sie die Textdatei z.B. als *Maximum3.vb* im Projektverzeichnis ab.

Kompilieren

Die Vorgehensweise entspricht exakt dem Abschnitt 1.2 im ersten Kapitel, Sie müssen also zunächst diverse Vorbereitungen treffen (Umgebungsvariable für den VB-Compiler hinzufügen, Datei *cmd.exe* in das Anwendungsverzeichnis kopieren), um bequem kompilieren zu können.

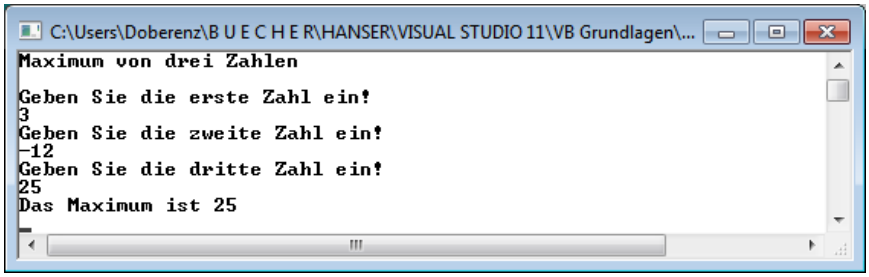
Geben Sie an der Kommandozeile ein:

```
vbc /t:exe Maximum3.vb
```

Haben Sie beim Eintippen des Quellcodes keine Fehler gemacht, so verläuft das Kompilieren anstandslos, im Projektverzeichnis finden Sie nun die Datei *Maximum3.exe* vor.

Test

Starten Sie *Maximum3.exe* durch Doppelklick!



Durch Drücken der *Enter*-Taste beenden Sie die Anwendung.

2.9.2 Vom Konsolen- zum Windows-Programm

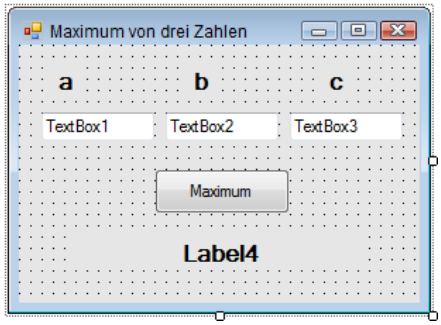
Eine Windows-Anwendung ist natürlich viel attraktiver als eine triste Konsolen-Applikation und schließlich wollen Sie ja zukünftig mit dem Komfort von Visual Studio anstatt mit einem simplen Texteditor arbeiten!

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 Basic*, 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 obiger Abbildung auf *Form1* und stellen auch hier bestimmte Eigenschaften (Beschriftungen) neu ein.

Quelltext

Durch einen Doppelklick auf *Button1* wird automatisch das Codefenster von *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 nie selbst eintippen, sondern immer durch die Visual Studio-Entwicklungsumgebung erzeugen lassen (siehe auch Einführungsbeispiele 1.7.1 und 1.7.2)!

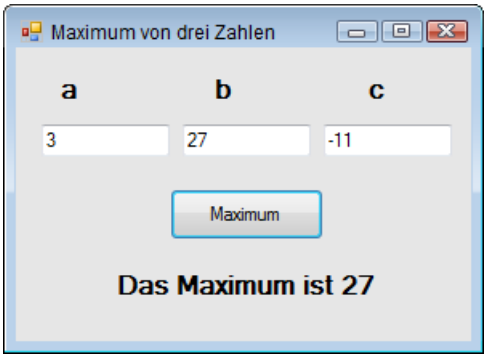
Füllen Sie den Körper des Eventhandlers mit den erforderlichen Anweisungen aus, so dass der komplette Eventhandler schließlich folgendermaßen aussieht:

```
Private Sub Button1_Click(sender As Object, e As EventArgs)  Handles Button1.Click
        Dim a, b, c, max As Integer           ' Variablendeklaration
        a = CInt(TextBox1.Text)               ' Eingabe a
        max = a                               ' Initialisieren von max
        b = CInt(TextBox2.Text)               ' Eingabe b
        If b > max Then max = b               ' Bedingung
        c = CInt(TextBox3.Text)               ' Eingabe c
        If c > max Then max = c               ' Bedingung
        Label4.Text = "Das Maximum ist " + max.ToString    ' Ergebnisausgabe
End Sub
```

Beim Vergleich mit der Konsolenanwendung erkennen Sie, dass die Programmierung von Ein- und Ausgabe deutlich einfacher geworden sind!

Test

Mittels *F5*-Taste kompilieren und starten Sie das Programm:



Bemerkungen

Neben dem attraktiveren Outfit 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.

2.9.3 Schleifenanweisungen kennen lernen

Visual Basic ist an Schleifenanweisungen reich gesegnet. Dieses Beispiel demonstriert Ihnen die prinzipielle Anwendung jedes Schleifentyps (siehe 2.6.2). Dabei soll zehnmal untereinander der Text "Viele Wege führen nach Rom!" in einer *ListBox* auszugeben werden, wobei acht verschiedene Schleifenkonstruktionen zur Anwendung kommen.

Oberfläche

Wie die folgende Laufzeitansicht zeigt, brauchen Sie neben einer *ListBox* auch eine ganze Menge von *Button*s (für jeden Schleifentyp einen und je einen zum Löschen des *ListBox*-Inhalts und zum Beenden des Programms).

Quellcode

Wir beginnen mit der altbekannten *For...Next*-Schleife:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim i As Integer
    For i = 1 To 10
```

```
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
        Next i
    End Sub
```

Do While...Loop-Schleife:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim i As Integer = 1
        Do While i <= 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop
    End Sub
```

Do Until...Loop-Schleife:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim i As Integer = 1
        Do Until i > 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop
    End Sub
```

Do...Loop While-Schleife:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim i As Integer = 1
        Do
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            i += 1
        Loop While i <= 10
    End Sub
```

Do...Loop Until-Schleife:

```
    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        Dim i As Integer = 1
        Do
           ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
           i += 1
        Loop Until i > 10
    End Sub
```

Do...Loop-Schleife mit *Exit Do*-Abbruch:

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        Dim i As Integer = 1
        Do
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
            If i = 10 Then Exit Do
            i += 1
        Loop
```

```
    End Sub
```

While...End While-Schleife:

```
    Private Sub Button7_Click(sender As Object,e As EventArgs) Handles Button7.Click
        Dim i As Integer = 1
        While i <= 10
            ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
          i += 1
        End While
    End Sub
```

Obwohl es sich hier um einen völlig anderen Schleifentyp handelt, soll vergleichsweise noch die Funktion der *For Each*-Schleife demonstriert werden:

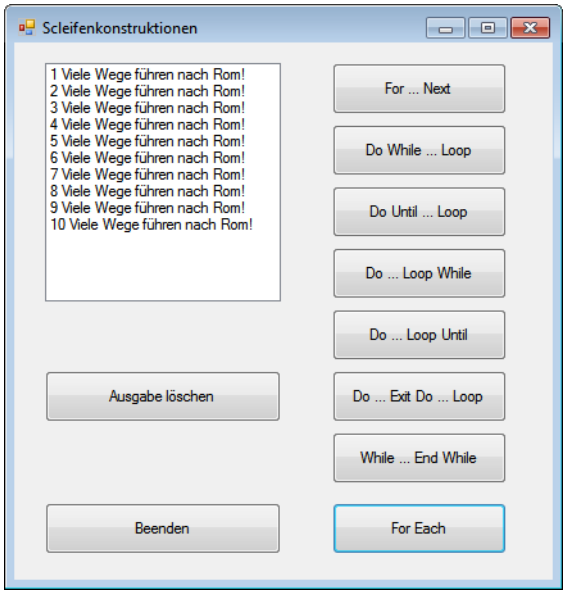
```
Private Sub Button8_Click(sender As Object, e As EventArgs) Handles Button8.Click
    Dim zahlen = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    For Each i As Integer In zahlen
        ListBox1.Items.Add(i.ToString & " Viele Wege führen nach Rom!")
    Next
End Sub
```

Eher nebensächlich das Löschen des Inhalts der *ListBox*:

```
Pivate Sub Button9_Click(sender As Object, e As EventArgs) Handles Button9.Click
    ListBox1.Items.Clear()
End Sub
```

Test

Alle acht Schleifenvarianten sollten ein absolut identisches Ergebnis erzeugen:



HINWEIS: Die *For Each*-Schleife ist mit der *For ... Next*-Schleife vergleichbar. Diesmal zählt sie allerdings keine Variable hoch, sondern die Schleife greift auf alle Elemente einer Auflistung zu. Es ist allerdings nur Lesezugriff erlaubt.

2.9.4 Methoden überladen

In VB spricht man im klassischen Sinn noch von Funktionen (*Function*s) und Prozeduren (*Sub*s), je nachdem ob es sich um Methoden mit und ohne Rückgabewert handelt. Beschränkt man sich auf einen privaten Gültigkeitsbereich (Aufruf nur innerhalb einer Klasse), so spielen diese quasi die Rolle von "Unterprogrammen", sind also strenggenommen keine Methoden im Sinne der OOP. Trotzdem können auch sie 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 (*Structure*), *If ... ElseIf*-Statements, *RadioButton*-Auswahl ... 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 Laufzeitabbildung).

Quellcode

```
Public Class Form1
```

Die *Kugel*-Struktur kapselt Radius und spezifisches Gewicht einer Kugel:

```
    Private Structure Kugel
        Dim radius, sg As Double
    End Structure
```

Eine globale Variable dient lediglich als Rückgabewert für Variante 2:

```
    Private gew As Double
```

Die folgende Hilfsfunktion liest die Kugelwerte aus der Eingabemaske in eine Strukturvariable vom oben definierten *Kugel*-Typ:

```
    Private Function getKugel() As Kugel
        Dim kug As Kugel
```

Den Kugelradius zuweisen:

```
        kug.radius = Convert.ToDouble(TextBox1.Text) / 2
```

Das spezifische Gewicht zuweisen:

```
        If RadioButton1.Checked Then
            kug.sg = 1.4 ' Holz
        ElseIf RadioButton2.Checked Then
            kug.sg = 2.7      ' Alu
        ElseIf RadioButton3.Checked Then
            kug.sg = 3D       ' Glas
        ElseIf RadioButton4.Checked Then
            kug.sg = 7.87D    ' Eisen
        Else
            kug.sg = 11.3     ' Blei
        End If
        Return kug
    End Function
```

Variante 1 zeigt als erste Überladung eine normale Funktion zur Bestimmung des Kugelgewichts:

```
    Private Function KugelGewicht(ra As Double, sg As Double) As Double
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(ra, 3)
        Return sg * vol     ' Rückgabe des Gewichts als Funktionswert
    End Function
```

Der Aufruf:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim kug As Kugel = getKugel()
        Dim r As Double = kug.radius, sg = kug.sg      ' Radius und spez. Gewicht
        Dim gew As Double = KugelGewicht(r, sg)        ' Aufruf einer Funktion
        Label3.Text = gew.ToString("#,#0.000 Gramm")   ' lokale Variable gew
    End Sub
```

Variante ist eine Prozedur (Sub), die das Ergebnis der globalen Variablen *gew* direkt zuweist:

```
    Private Sub KugelGewicht(kug As Kugel)
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(kug.radius, 3)
        gew = kug.sg * vol
    End Sub
```

Der Aufruf:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim kug As Kugel = getKugel()
        KugelGewicht(kug)
        Label4.Text = gew.ToString("#,#0.000 Gramm")
    End Sub
```

Variante 3 ist ebenfalls eine Prozedur (Sub), hier aber wird das Gewicht über den Parameter *g* per Referenz zurückgegeben:

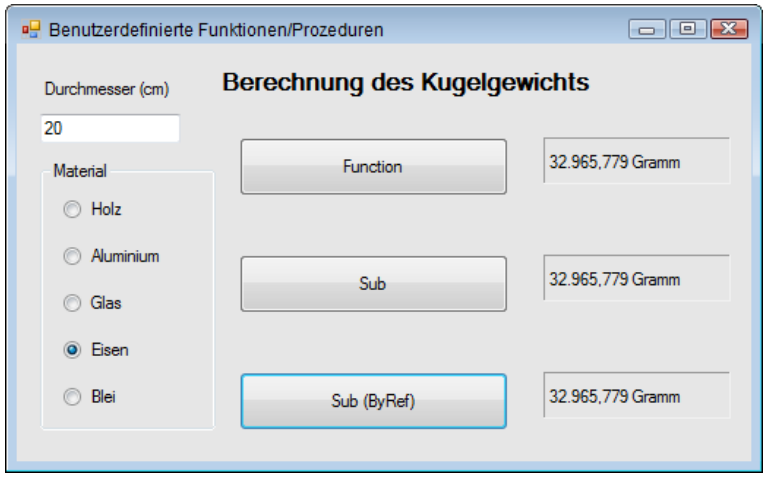
```
    Private Sub KugelGewicht(kug As Kugel, ByRef g As Double)
        Dim vol As Double = 4 / 3 * Math.PI * Math.Pow(kug.radius, 3)
        g = kug.sg * vol
    End Sub
```

Aufruf Variante 3:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim kug As Kugel = getKugel()
        Dim gew As Double
        KugelGewicht(kug, gew)
        Label5.Text = gew.ToString("#,#0.000 Gramm")
    End Sub
End Class
```

Test

Geben Sie das Material und den Durchmesser der Kugel ein und lassen Sie sich das Gewicht anzeigen. Alle drei Überladungen liefern identische Ergebnisse[1]:



2.9.5 Eine Iterationsschleife verstehen

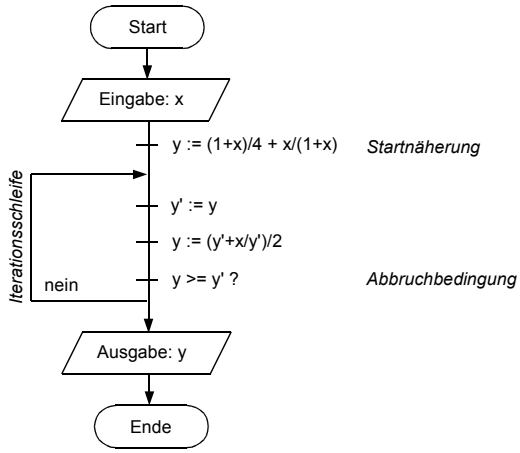
Eigentlich ist ein Computer ja "dumm" und erscheint nur dadurch "intelligent", dass er primitive Rechenoperationen in hoher Geschwindigkeit erledigt. Der pfiffige Programmierer kann dies ausnutzen, indem er dem Computer Aufgaben stellt, die sich nicht sofort, sondern nur durch schrittweises Ausprobieren lösen lassen. Typisch für diese Sorte von Aufgaben ist eine so genannte *Iterationsschleife*, die mit einer "über den Daumen gepeilten" *Startnäherung* beginnt und an deren Ende eine *Abbruchbedingung* überprüft wird. Wie viele Male die Schleife durchlaufen wird, kann nicht exakt vorausbestimmt werden.

[1] Kaum zu glauben aber wahr: Eine Eisenkugel mit nur 20 cm Durchmesser wiegt stolze 33 Kilogramm!

Ohne auf die mathematischen Grundlagen näher einzugehen, wollen wir in diesem Beispiel eine Iterationsschleife für das Ziehen der Quadratwurzel demonstrieren. Wir verzichten also auf die *Sqrt*-Funktion, wie sie standardmäßig von der *Math*-Klasse bereitgestellt wird, und programmieren stattdessen eine eigene Lösung.

Programmablaufplan

Obwohl in der objekt- und ereignisorientierten Programmierung der PAP völlig aus der Mode gekommen ist, eignet er sich nach wie vor gut zur Veranschaulichung von Iterationszyklen.



Oberfläche

Für die Eingabe findet eine *TextBox*- und für die Ausgabe eine *Label*-Komponente Verwendung. Die Iterationsschleife starten wir mit einem *Button*. Eine *ListBox* ist nicht unbedingt erforderlich, aber wir sind ja neugierig und wollen auch die Zwischenergebnisse betrachten (siehe Laufzeitabbildung).

Quellcode

Die programmtechnische Umsetzung des obigen PAP führt (in Verbindung mit dem Code für die Ein- und Ausgabe und für die Anzeige von Zwischenergebnissen) zu folgender Lösung:

```
Public Class Form1
```

Die Schaltfläche *Quadratwurzel>>*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Einige Variablen deklarieren und den Eingabewert explizit in einen *Double*-Wert konvertieren:

```
        Dim x, y, ya As Double
        x = Convert.ToDouble(TextBox1.Text)
```

Die Quadratwurzel darf nur aus positiven Zahlen gezogen werden:

```
        If x > 0 Then
```

Inhalt der *ListBox* löschen:

```
            ListBox1.Items.Clear()
```

Die Startnäherung:

```
            y = (1 + x) / 4 + x / (1 + x)
```

Wie geschaffen für unsere Iterationsschleife ist die *Do...Loop Until*-Anweisung:

```
            Do
                ya = y
                y = (ya + x / ya) / 2
```

Das Zwischenergebnis anzeigen:

```
                ListBox1.Items.Add(y.ToString)
```

Die Abbruchbedingung prüfen:

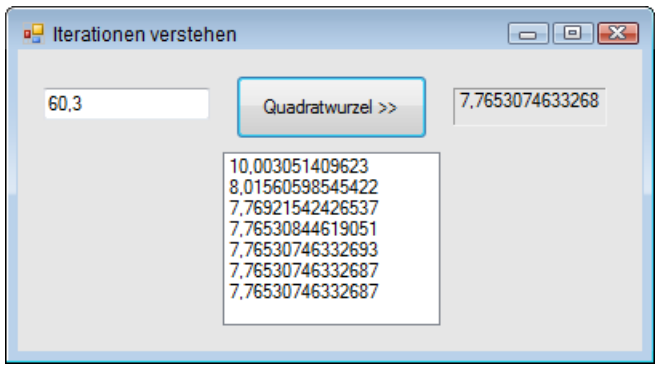
```
            Loop Until y >= ya
```

Das Endergebnis anzeigen:

```
            Label1.Text = y.ToString
        Else
            Label1.Text = "Bitte geben Sie einen positiven Wert ein!"
        End If
    End Sub
End Class
```

Test

Die Zwischenergebnisse nähern sich schrittweise der endgültigen Lösung. Sie werden feststellen, dass circa fünf bis sieben Iterationen notwendig sind, um die Abbruchbedingung zu erfüllen, d h., die Quadratwurzel in einer für *Double*-Zahlen ausreichenden Genauigkeit zu ermitteln:



Ergänzung

Der fortgeschrittene Programmierer wird obigen Code – besonders im Hinblick auf seine Wiederverwendbarkeit – in eine Funktion *qWurzel* verpacken. Da die Testphase abgeschlossen ist, kann auch auf die Anzeige der Zwischenergebnisse in der *ListBox* verzichtet werden:

```
Public Function qWurzel(x As Double) As Double
    Dim y, ya As Double
    y = (1 + x) / 4 + x / (1 + x)   ' Startnäherung
    Do
        ya = y
        y = (ya + x / ya) / 2
    Loop Until y >= ya              ' Abbruchbedingung
    Return y
End Function
```

Der Aufruf gestaltet sich nun wesentlich übersichtlicher:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim x As Double = Convert.ToDouble(TextBox1.Text)
    If x > 0 Then
        Label1.Text = qWurzel(x).ToString
    Else
        Label1.Text = "Bitte geben Sie einen positiven Wert ein!"
    End If
End Sub
```

2.9.6 Anwendungen von C# nach Visual Basic portieren

Sie sind C#-Programmierer und wollen zu Visual Basic wechseln? Oder Sie haben einen in C# geschriebenen Quellcode und wollen diesen nach Visual Basic übersetzen? Das vorliegende Rezept soll Ihnen den Umstieg erleichtern.

Leider können wir Ihnen für eine automatisierte Durchführung keine Patentlösung geben. Im Internet werden zwar diverse Übersetzungshilfen angeboten, sie sind allerdings teilweise so mit Mängeln behaftet, dass wir sie Ihnen beim besten Willen nicht empfehlen wollen.

Am schnellsten und zuverlässigsten geht das Portieren immer noch in "Handarbeit":

- Öffnen Sie das zu übersetzende C#-Projekt mit Visual Studio.
- Laden Sie eine zweite Instanz von Visual Studio und öffnen Sie ein neues Visual Basic-Projekt.
- Kopieren Sie über die Windows-Zwischenablage die visuellen Komponenten (keine Formulare!) von der C#- in die VB-Entwicklungsumgebung.
- Kopieren Sie abschnittsweise den C#-Code in das VB-Codefenster und ersetzen Sie dort Schritt für Schritt die C#- durch die VB-Syntax.

Die folgenden Ausführungen sollen Ihnen dabei die Arbeit erleichtern.

Die augenfälligsten Unterschiede

Beim Vergleich beider Programmiersprachen stechen zunächst die folgenden Unterschiede ins Auge:

- **Abschluss einer Befehlszeile**
  Während in Visual Basic der Zeilenumbruch eine Befehlszeile abschließt, übernimmt in C# das Semikolon diese Funktion, sodass sich Anweisungen über mehrere Zeilen erstrecken können!
- **Groß-/Kleinschreibung**
  Hier dürfte es kaum Probleme geben, denn in Visual Basic spielt die Groß-/Kleinschreibung keine Rolle, C# hingegen ist eine casesensitive Sprache, die in "Oberlehrermanier" peinlichst zwischen Groß- und Kleinschreibung unterscheidet.
- **Kommentare**
  In C# sind auch mehrzeilige, in VB hingegen nur einzeilige, Kommentare möglich.
- ***this* wird zu *Me***
  Als Bezug auf die eigene Klasseninstanz benutzt man unter C# das Schlüsselwort *this*, unter VB aber *Me*.
- **Gültigkeitsbereiche**
  Während die Gültigkeitsbereiche der lokalen Variablen sich in C# eindeutig aus der durch die geschweiften Klammern eingegrenzten Blockstruktur ergeben, kann man in VB die Blöcke nur indirekt aus der Struktur der Anweisungen ableiten.
- **Namespaces**
  Während ein Namensraum unter VB mit *Imports* eingebunden wird, erfolgt dies unter C# mit *using*.

Ehe wir weiter ins Detail gehen, zeigt die folgende Tabelle eine zeilenweise Gegenüberstellung von zwei kompletten Programmen (Konsolenanwendungen).

HINWEIS: Den Quellcode für beide Programme finden Sie in den Beispieldaten!

C#

```
using System;      // Namespace einbinden
class Program      /* Das Programm ist eine
                      Konsolenanwendung */
{
  static void Main(string[] args)
  {
    char c;
    Console.WriteLine("Umrechnung Euro-Dollar");
    do
    {
        float kurs, euro, dollar;
        Console.Write("Kurs 1 : ");
        kurs = Convert.ToSingle(Console.ReadLine());
        Console.Write("Euro: ");
```

VB

```
Imports System      ' Namespace einbinden
Module Module1      ' Das Programm ist eine
                    ' Konsolenanwendung

Sub Main()

    Dim c As Char
    Console.WriteLine("Umrechnung Euro-Dollar")
    Do

        Dim kurs, euro, dollar As Single
        Console.Write("Kurs 1 : ")
        kurs = CSng(Console.ReadLine())
        Console.Write("Euro: ")
```

C#	VB

`        euro = Convert.ToSingle(Console.ReadLine());`
`        dollar = euro * kurs;`
`        Console.WriteLine("Sie erhalten " +`
`                    dollar.ToString("0.00 Dollar"));`
`        Console.Write("Programm beenden? (j/n)");`
`        string s = Console.ReadLine(); c = s[0];`
`    } while(c != 'j');`
`  }`
`}`
	`        euro = CSng(Console.ReadLine())`
	`        dollar = euro * kurs`
	`        Console.WriteLine("Sie erhalten " &`
	`        dollar.ToString("0.00 Dollar"))`
	`        Console.Write("Programm beenden? (j/n)")`
	`        c = CChar(Console.ReadLine())`
	`      Loop While c <> "j"`
	`   End Sub`
	`End Module`

Datentypen

C#-Datentyp	VB.NET-Datentyp	.NET-CLR-Typ
byte	*Byte*	*System.Byte*
sbyte	*SByte*	*System.SByte*
short	*Short*	*System.Int16*
ushort	*UShort*	*System.UInt16*
int	*Integer*	*System.Int32*
uint	*UInteger*	*System.UInt32*
long	*Long*	*System.Int64*
ulong	*ULong*	*System.UInt64*
float	*Single*	*System.Single*
double	*Double*	*System.Double*
decimal	*Decimal*	*System.Decimal*
char	*Char*	*System.Char*
bool	*Boolean*	*System.Boolean*
string	*String*	*System.String*
object	*Object*	*Object*

Operatoren

C#	VB	C#	VB	C#	VB
`=  +  -`	`=  +  -`	`&`	`And`	`==`	`=`
`++`		`|`	`Or`	`<  >`	`<  >`
`*  /`	`*  /`	`^`	`Xor`	`<=  >=`	`<=  >=`
	`\`	`&&`	`AndAlso`	`!=`	`<>`
`%`	`Mod`	`||`	`OrElse`		
	`^`	`!`	`Not`		

Verzweigungen

VB	C#

`If zensur = 1 Then`
`  Label1.Text =  "Sehr gut!"`
`ElseIf zensur = 2 Then`
`  Label1.Text = "Gut"`
`ElseIf zensur = 3`
`  Label1.Text = "Befriedigend"`
`Else`
`  Label1.Text = "Nicht erlaubte Zensur!"`
`End If`
	`if (zensur == 1)`
	` label1.Text =  "Sehr gut!";`
	`else if (zensur == 2)`
	` label1.Text =  "Gut";`
	`else if (zensur == 3)`
	` label1.Text =  "Befriedigend";`
	`else`
	`  label1.Text = "Nicht erlaubte Zensur!";`

`Select Case zensur`
`  Case 1: Label1.Text = "Sehr gut"`
`  Case 2: Label1.Text = "Gut"`
`  Case 3: Label1.Text = "Mmmm"`
`  Case Else`
`  Label1.Text = "????"`
`End Select`
	`switch (zensur)`
	`{ case 1: label1.Text = "Sehr gut"; break;`
	`  case 2: label1.Text = "Gut"; break;`
	`  case 3: label1.Text = "Mmmm"; break;`
	`  default:`
	`  label1.Text = "????"; break;`
	`}`

`Try`
`  c = a / b`
`  Me.Text = c.ToString`
`Catch ex As Exception`
`  MessageBox.Show(ex.Message)`
`End Try`
	`try`
	`{`
	`  c = a / b;`
	`  this.Text = c.ToString();`
	`}`
	`catch(Exception ex)`
	`{`
	`  MessageBox.Show(ex.Message);`
	`}`

Schleifen

VB	C#

`For i As Integer = 1 To 10`
`  ListBox1.Items.Add("Hallo!")`
`Next i`
	`for (int i = 1; i <= 10; i++)`
	`  listBox1.Items.Add(" Hallo!");`

`Do While i  <= 10`
`  ListBox1.Items.Add("Hallo!")`
`  i += 1`
`Loop`
	`while (i <= 10)`
	`{ listBox1.Items.Add("Hallo!");`
	`  i++;`
	`}`

`Do`
` ListBox1.Items.Add("Hallo!")`
` i +=1`
`Loop While i <=10`
	`do`
	`{ listBox1.Items.Add("Hallo!");`
	`  i ++;`
	`} while (i <= 10);`

Arrays

VB	C#
`Dim A(100) As Double` `A(0) = 12.5`	`double[] A = new double[100];` `A[0] = 12.5;`

`Dim feld As Double`
`For Each feld In A`
`  MessageBox.Show(feld.ToString)`
`Next feld`
	`foreach (double feld in A)`
	`{`
	`  MessageBox.Show(feld.ToString());`
	`}`

Strukturen

VB	C#

`Structure Person`
`  Public vorName, nachName As String`
`  Public alter As Integer`
`End Structure`
	`struct Person`
	`{ public string vorName, nachName;`
	`  public int alter;`
	`}`

Enumerationen

VB	C#

`Enum erstesQuartal As Byte`
`  JANUAR = 1`
`  FEBRUAR`
`  MÄRZ`
`End Enum`
	`enum erstesQuartal:  byte`
	`{JANUAR = 1,`
	` FEBRUAR,`
	` MÄRZ`
	`}`

- Arrays sind in .NET alle dynamisch und beginnen grundsätzlich mit dem Index 0, dieser ist in C# in eckige, in VB hingegen in runde Klammern einzuschließen.
- In VB ist einfaches Umdimensionieren von Arrays zur Laufzeit mittels *ReDim*-Anweisung möglich.

Funktionen, Prozeduren, Methoden

VB	C#

`Function Kugel(ra As Double,`
`  sg As Double) As Double`
`  Dim vol As Double =`
`  4  / 3 * Math.Pi * Math.Pow(ra, 3)`
`  Return sg * vol`
`End Function`
	`double Kugel(double ra, double sg)`
	`{`
	`  double vol =`
	`  4 / 3.0 * Math.PI * Math.Pow(ra, 3);`
	`  return(sg * vol);`
	`}`

VB	C#

`Sub Kugel(ra As Double,`
`  sg As Double)`
` Dim vol As Double =`
`  4 / 3 * Math.Pi * Math.Pow(ra, 3)`
` gew = sg * vol`
`End Sub`
	`void Kugel(double ra, double sg)`
	`{`
	`  double vol =`
	`  4 / 3F * Math.PI * Math.Pow(ra, 3);`
	`  gew = sg * vol;`
	`}`

`Declare Function GetDC Lib "user32" _`
`  Alias "GetDC" (hwnd As Integer) _`
`  As Integer`
` ...`
`Dim dc As Integer = GetDC(0)`
	`[DllImport("user32.dll")]`
	`static extern int  GetDC(int  hwnd);`
	
	`...`
	`int dc = GetDC(0);`

HINWEIS: VB unterscheidet zwischen Funktionen und Prozeduren, C# kennt nur Methoden mit und ohne (*void*) Rückgabewert.

Klassendefinition

VB	C#

`Public Class CKunde`
 	
`  Public Shared Const mwst As Double = 0.19`
`  Private _anrede, _name As String`
 	
`  Public Sub New(anr As String,`
`  nam As String)`
`  _anrede = anr: _name = nam`
`  End Sub`
 	
`  Public Property name() As String`
`  Get`
`  Return _name`
`  End Get`
`  Set(Value As String)`
`  _name = Value`
`  End Set`
`  End Property`
`  Public Function adresse() As String`
`  Dim s As String = _anrede & " " & _name`
`  Return s`
`  End`
 	
`End Class`
	`public class CKunde`
	`{`
	`  public static const double mwst = 0.19;`
	`  private string _anrede, _name;`
	
	`  public CKunde(string anr, string nam)`
	`  {`
	`  _anrede = anr;  _name = nam;`
	`  }`
	
	`  public string name`
	`  {`
	`  get`
	`  {return(_name);}`
	`  set`
	`  {_name = value;}`
	`  }`
	
	`  public string adresse()`
	`  {`
	`  string s = _anrede + " " + _name;`
	`  return(s);`
	`  }`
	` }`

Erzeugen eines Objekts

VB	C#
`Private kunde1 As CKunde` `kunde1 = New CKunde("Frau", "Maus")`	`private CKunde kunde1;` `kunde1 = new CKunde1("Frau", "Maus");`
`Private kunde1 As New CKunde("Frau", "Maus")`	`private CKunde kunde1 = new CKunde("Frau", "Maus");`

Ereignis definieren

VB	C#

`Public Event ereignis(`
` sender As Object, e As String)`
`...`
`RaiseEvent ereignis(Me, "Hallo")`
	`public delegate void EreignisTyp(`
	`object sender, string e);`
	
	`public event EreignisTyp ereignis;`
	`...`
	`ereignis(this, "Hallo");`

Ereignis verwenden

VB	C#

`Private WithEvents kunde1 As CKunde`
` kunde1 = New CKunde`
 	
` AddHandler kunde1.ereignis,`
`AddressOf Me.ereignisHandler`
` ...`
` Private Sub ereignisHandler(`
`  sender As Object, e As String)`
`  ...`
` End Sub`
	`private CKunde kunde1;`
	` kunde1 = new CKunde();`
	
	` kunde1.ereignis += new`
	`  CKunde.EreignisTyp(this.ereignisHandler);`
	` ...`
	` private void ereignisHandler(`
	`object sender, string e)`
	` {`
	`  ...`
	` }`

HINWEIS: Unter VB ist auch das (statische) Zuweisen eines Eventhandlers mittels Handles-Klausel möglich.

Kapitel 3

OOP-Konzepte

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.

Visual Basic erlaubt es Ihnen aber, 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 (Event-Handler) 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 Visual Basic im Überblick.

3.1 Strukturierter versus objektorientierter Entwurf

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.

3.1.1 Was bedeutet 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 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.

Dass VB 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 VB 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. anstatt mit Methoden mit Funktionen und Prozeduren und anstatt mit Klassen mit strukturierten Datentypen (*Structure*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt.

Anstatt 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.

3.1.2 Was heißt objektorientierte Programmierung?

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.

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.

Das Konzept der objektorientierten Programmierung (OOP) überwindet den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

3.2 Grundbegriffe der OOP

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung zunächst allgemein, d h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

3.2.1 Objekt, Klasse, Instanz

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.

Eine *Klasse*[1] 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: In VB haben Sie grundsätzlich die Möglichkeit, zwischen Klassen und Strukturen zu wählen. Letztere wurden bereits im Sprachkapitel (Abschnitt 2.7.2) 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 beschäftigen.

[1] Oft wird anstatt "Klasse" mit völlig gleichwertiger Bedeutung auch der Begriff "Objekttyp" (oder auch "Typ") verwendet.

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: Objekt, Klasse, 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.

3.2.2 Kapselung und Wiederverwendbarkeit

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.

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.

3.2.3 Vererbung und Polymorphie

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 (auch oft als Superklasse bezeichnet) 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".

Visual Basic unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden.

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: Vererbung

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.

3.2.4 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.
Friend	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.

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
Public	Unbeschränkt.
Friend	Innerhalb der aktuellen Anwendung wie *Public*, sonst wie *Private*.
Protected	Wie *Private*, Mitglieder dürfen aber auch in abgeleiteten Klassen verwendet werden.
Protected Friend	Innerhalb der aktuellen Anwendung wie *Protected*, sonst wie *Private*.
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 3.9 (Vererbung und Polymorphie).

3.2.5 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel mit den Strukturen (siehe Abschnitt 2.7.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen (der wesentliche Unterschied ist, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind). Auch im Aufbau von Funktionen bzw. Methoden sollte sich der Lernende auskennen (Abschnitt 2.8).

Im Unterschied zu einer Struktur (Schlüsselwort *Structure*) wird eine Klasse mit dem Schlüsselwort *Class* deklariert. Hier die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer Class Bezeichner
        ' ... Felder
        ' ... Konstruktoren
        ' ... Eigenschaften
        ' ... Methoden
        ' ... Ereignisse
    End Class
```

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.

VB
```
Public Class CKunde
   Private _anrede As String          '    Feld
   Private _name As String            '    dto.

   Public Sub New(anr As String, nam As String)    ' Konstruktor
      _anrede = anr
```

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

```
      _name = nam
   End Sub
   Public Property name() As String                ' Eigenschaft
      Get
         Return _name
      End Get
      Set(value As String)
         _name = value
      End Set
   End Property

   Public Function adresse() As String             ' Methode
       Dim s As String = _anrede & " " & _name
       Return s
   End Function
End Class
```

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.

3.3 Ein Objekt erzeugen

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 *Nothing* 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 *Nothing*)
- 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.

3.3.1 Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

Variante 1 erfordert zwei Schritte:

SYNTAX:
```
Modifizierer myObject As Klasse
myObjekt = New Klasse(Parameter)
```

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

VB
```
Private kunde1 As CKunde                    ' 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:
```
Modifizierer myObject = New Klasse()
```

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

VB
```
Private kunde1 As 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 (*Imports*-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].

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

VB
```
Private kunde1 As CKunde
Try
   kunde1 = New CKunde()
Catch e As Exception
   MessageBox.Show(ex.Message)
End Try
```

[1] Aus Platz- und Bequemlichkeitsgründen halten sich auch die Autoren nicht immer an diese Empfehlung.

3.3.2 Klassische Initialisierung

Anstatt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben. Zunächst ein Beispiel ohne eigenen Konstruktor, wobei der parameterlose Standardkonstruktor zum Einsatz kommt.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

VB
```
Dim kunde1 As New CKunde()
kunde1.anrede = "Frau"
kunde1.name = "Müller"
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

VB
```
Dim kunde1 As New CKunde("Frau", "Müller")
```

HINWEIS: Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.8.1.

3.3.3 Objekt-Initialisierer

Man kann ein Objekt auch dann erzeugen und seine Eigenschaften (keine privaten Felder!) initialisieren, wenn es dazu keinen Konstruktor gibt.

BEISPIEL 3.9: Das Vorgängerbeispiel mit Objekt-Initialisierer

VB
```
Private kunde1 As New CKunde With {.anrede = "Frau", .name = "Müller"}
```

HINWEIS: Mehr zu Objekt-Initialisierern siehe Abschnitt 3.8.2!

3.3.4 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.

VB
```
kunde1.Guthaben = 10
Label1.Text = kunde1.adresse()
```

3.3.5 Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *Nothing* setzen. Vorher können Sie (müssen aber nicht) die *Dispose*-Methode des Objekts aufrufen, vorausgesetzt, Sie haben Sie auch implementiert.

BEISPIEL 3.11: Der *kunde1* wird entfernt.

VB
```
kunde1.Dispose()
kunde1 = Nothing
```

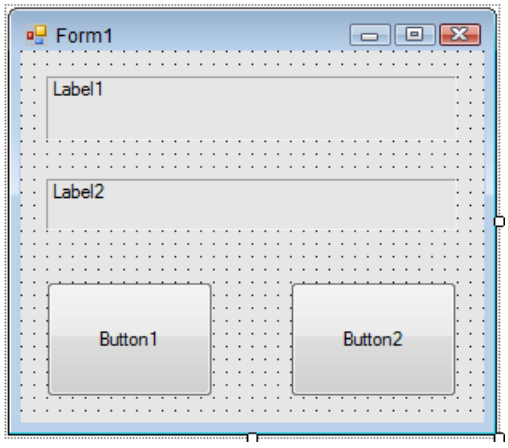
HINWEIS: Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird.

3.4 OOP-Einführungsbeispiel

Raus aus dem muffigen Hörsaal, lassen Sie uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

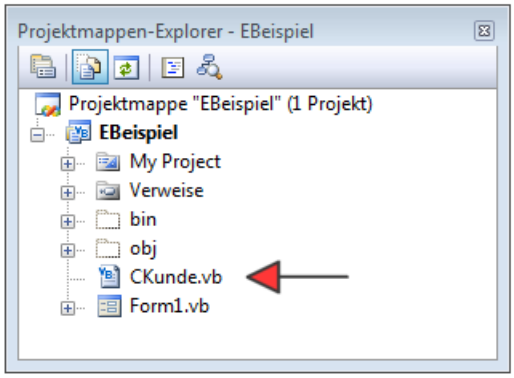
3.4.1 Vorbereitungen

- 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.vb* ein und klicken "Hinzufügen".

Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



HINWEIS: 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.vb*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

3.4.2 Klasse definieren

Tragen Sie in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
Public Class CKunde
    Public Anrede As String             ' einfache Eigenschaften
    Public Name As String               '      dto.
    Public PLZ As Integer               '      dto.
    Public Ort As String                '      dto.
    Public Stammkunde As Boolean        '      dto.
    Public Guthaben As Decimal          '      dto.

    Public Function getAdresse() As String              ' erste Methode
        Dim s As String = Anrede & " " & Name & vbCrLf & PLZ.ToString & " " & Ort
        Return s
    End Function

    Public Sub addGuthaben(betrag As Decimal)     ' zweite Methode
        If Stammkunde Then Guthaben += betrag
    End Sub
End Class
```

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* (eine Funktion und eine Prozedur). Die Funktion *getAdresse()* liefert als Rückgabewert die komplette Anschrift des Kunden.
- Die Prozedur (*Sub*) *addGuthaben()* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *Guthaben*-Felds bei jedem Aufruf um einen bestimmten Betrag.

3.4.3 Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Zu Beginn deklarieren Sie eine Objektvariable *kunde1*:

```
Private kunde1 As CKunde            ' Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen sein *Click*-Ereignis wie folgt:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1 = New CKunde()          ' Objekt erzeugen (instanziieren)
        With Kunde1                    ' Objekt initialisieren:
            .Anrede = "Herr"
            .Name = "Müller"
            .PLZ = 12345
            .Ort = "Berlin"
            .Stammkunde = True
        End With
    End Sub
```

3.4.4 Objekt verwenden

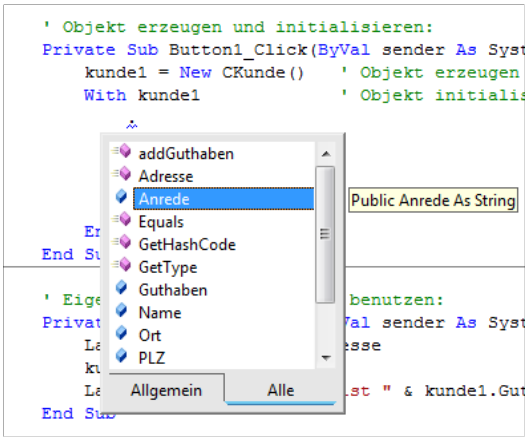
Hinterlegen Sie nun den rechten Button mit der Beschriftung "Eigenschaften und Methoden verwenden" wie folgt:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Label1.Text = Kunde1.getAdresse    ' erste Methode aufrufen
        Kunde1.addGuthaben(50D)            ' zweite Methode aufrufen
        Label2.Text = "Guthaben ist " & Kunde1.Guthaben.ToString("C")    ' Eigenschaft lesen
    End Sub
```

3.4.5 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 doppelt auf den gewünschten Eintrag klicken.

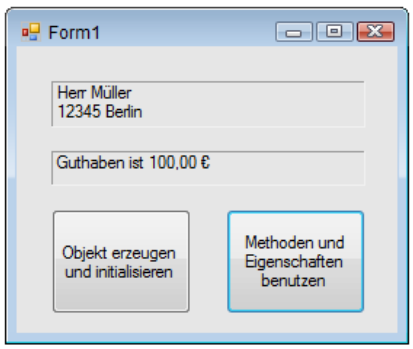


Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *Public*-Modifizierer vergessen wurde.

3.4.6 Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes VB-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.



3.4.7 Warum unsere Klasse noch nicht optimal ist

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 Visual Basic 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.
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht).
- 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, können und wollen wir aber nicht weiter aufschieben und es gleich im folgenden Abschnitt behandeln.

3.5 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.

3.5.1 Eigenschaften 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 *Get*- und für den Schreibzugriff eine *Set*-Zugriffsmethode. Während die *Get*-Methode ihren Rückgabewert über *Return* liefert, erhält die *Set*-Methode den zu schreibenden Wert über den Parameter *value*.

```
SYNTAX:  [Public|Friend|Protected] Property Eigenschaftsname As Type
                Get
                     ' hier Lesezugriff (Wert=priv.Felder) implementieren
                     Return Wert
                End Get
                Set(value As Type)
                      ' hier Schreibzugriff (priv.Felder=value) implementieren
                 End Set
           End Property
```

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.

```
Public Class CKunde
    Private _anrede As String        '    privates Feld
    Private _name As String          '      dto.
    ...
```

Der Schreibzugriff auf die Eigenschaft *Anrede* soll so kontrolliert werden, dass nur die Werte "Herr" oder "Frau" zulässig sind. Geben Sie die erste Zeile der *Property*-Deklaration ein, so generiert Visual Studio automatisch den kompletten Rahmencode:

```
Public Property Anrede As String
    Get

    End Get
    Set(value As String)

    End Set
End Property
```

Sie brauchen dann nur noch den Lese- und den Schreibzugriff zu implementieren, sodass die komplette Eigenschaftsdefinition schließlich folgendermaßen aussieht:

```
    Public Property Anrede() As String
        Get
            Return _anrede
        End Get
        Set(value As String)
            If (value = "Herr") Or (value = "Frau") Then
                _anrede = value
            Else
```

```
            MessageBox.Show("Die Anrede '" & value & "' ist nicht zulässig!",
                                            "Fehler bei der Eingabe!")
        End If
    End Set
End Property
```

Beim Implementieren der Eigenschaft *Name* machen wir es uns etwas einfacher. Hier soll uns die einfache Kapselung genügen (es gibt also keinerlei Zugriffskontrolle):

```
Public Property Name() As String
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property
```

Zugriff

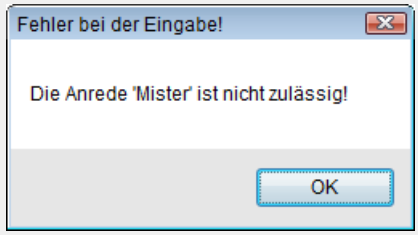
Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *Get*- bzw. *Set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

VB

```
Dim kunde1 As 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.
- 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 bei der *Name*-Eigenschaft nur wegen des Lerneffekts betrieben.

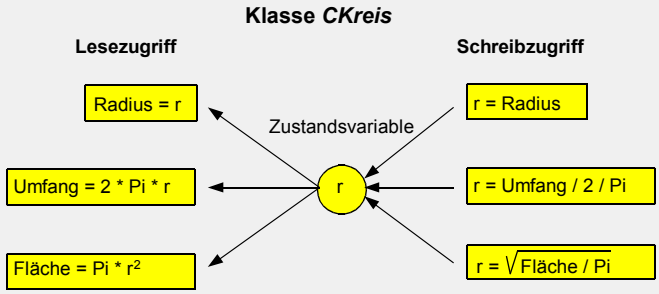
3.5.2 Eigenschaften mit Zugriffsmethoden kapseln

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.

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

VB

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 r As Double                 ' die einzige Zustandsvariable
```

Die Eigenschaft *Radius*:

```
    Public Property Radius() As String
        Get
            Return r.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = CDbl(value)
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * r).ToString("#,#0.00")
        End Get
        Set(value As String)
```

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

```
            If value <> String.Empty Then
                r = CDbl(value) / 2 / Math.PI
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                r = 0
            End If
        End Set
    End Property
End Class
```

Das komplette Programm finden Sie unter

► 3.12.1 Eigenschaften sinnvoll kapseln

3.5.3 Lese-/Schreibschutz für Eigenschaften

Es kommt häufig vor, dass bestimmte Eigenschaften nur gelesen oder nur geschrieben werden dürfen (*ReadOnly* bzw. *WriteOnly*). Um diese Art der Zugriffsbeschränkung zu realisieren, ist keinerlei 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 3.14: Lese-/Schreibschutz für Eigenschaften

VB

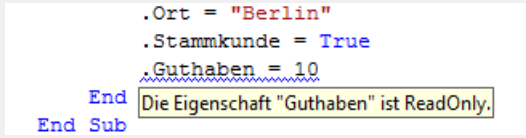
In unserer *CKunde*-Klasse soll das Guthaben für den direkten Schreibzugriff gesperrt werden. Das klingt logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
Public Class CKunde
    ...
     Public ReadOnly Property Guthaben() As Decimal
        Get
```

BEISPIEL 3.14: Lese-/Schreibschutz für Eigenschaften

```
            Return _guthaben
        End Get
    End Property
    ...
End Class
```

Bereits in der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:



3.5.4 Statische Eigenschaften

Mitunter gibt es 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 Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *Shared* deklariert.

Außer dem *Shared*-Schlüsselwort gibt es beim Deklarieren keine Unterschiede zu den normalen Instanzeneigenschaften.

Statische Eigenschaften können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden muss! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.15: Die Klasse *CKunde* soll zusätzlich eine "einfache" Eigenschaft *Rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

VB
```
Public Class CKunde
    ...
    Public Shared Rabatt As Double
    ...
End Class
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.16: Allen Kunden wird ein Rabatt von 15% zugewiesen.

VB
```
CKunde.Rabatt = 0.15
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre globalen Variablen zu verzichten, mit denen sie Werte zwischen verschiedenen Programmmodulen ausgetauscht haben. Genau hier bieten sich statische Eigenschaften an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

3.5.5 Selbst implementierende Eigenschaften

Bei sehr einfachen Eigenschaften (vergleichbar mit denen, die Sie bislang "unsauber" als *Public*-Felder deklariert haben) können Sie seit VB 2010 so genannte *Auto-implemented Properties* verwenden. Der VB-Compiler generiert im Hintergrund für Sie die entsprechenden *Get*- und *Set*-Zugriffsmethoden und erzeugt außerdem ein privates Feld, um den Wert der Eigenschaft zu speichern.

Wie viel lästige Schreibarbeit Sie sparen können, soll das folgende Beispiel verdeutlichen.

BEISPIEL 3.17: Die folgende Deklaration einer selbst implementierenden Eigenschaft

VB
```
Property Ort As String = "München"
```

ist äquivalent zu

```
Private _Ort As String = "München"                  ' backing field
Property Ort As String
    Get
        Return _Ort
    End Get
    Set(value As String)

        _Ort = value
    End Set
End Property
```

Wie Sie sehen, kann der Eigenschaft auch ein Standardwert zugewiesen werden. Der Name des automatisch angelegten "backing fields" entspricht dem Namen der Eigenschaft mit vorangestelltem Unterstrich.

HINWEIS: Achten Sie auf Namenskonflikte, die entstehen können, wenn Sie eigene Felder zur Klasse hinzufügen, die auch mit einem Unterstrich (_) beginnen!

Komplette Eigenschaftsdeklarationen lassen sich in einer einzigen Zeile erledigen, wie es die folgenden Beispiele zeigen.

BEISPIEL 3.18: Einige selbst implementierende Eigenschaften

VB
```
Public Property FullName As String
Public Property FullName As String = "Max Muster"
Public Property ID As New Guid()
Public Property ErstesQuartal As New List(Of String) From {"Januar", "Februar", "März"}
```

Seit VB 2015 gibt es auch selbst implementierende ReadOnly-Eigenschaften. Wie bei normalen Auto-Properties können Sie jetzt auch Werte zu ReadOnly-Properties zuweisen und das auch im Konstruktor.

BEISPIEL 3.19: ReadOnly-Eigenschaften

VB
```
Class CAutor
   Public ReadOnly Property Tags As New List(Of String)
   Public ReadOnly Property Name As String = ""
   Public ReadOnly Property Datei As String

   Sub New(datei As String)
      Me.Datei = datei
   End Sub
End Class
```

3.6 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.6.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben Sie 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 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

VB
```
Public Class CKunde
```

Private Variablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
    Private _ort As String
    Private _stammkunde As Boolean
```

BEISPIEL 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

```
    Private _guthaben As Decimal
```

Öffentliche Methoden:

```
Public Function getAdresse() As String
        Dim s As String = _anrede & " " & _name & vbCrLf & _plz.ToString & " " & _ort
        Return s
End Function

Public Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
End Sub
```

Der Aufruf:

```
Private kunde1 As New CKunde()
...
Label1.Text = kunde1.getAdresse         ' erste Methode (Funktion) aufrufen
kunde1.addGuthaben(50)                  ' zweite Methode (Prozedur) aufrufen
```

3.6.2 Überladene Methoden

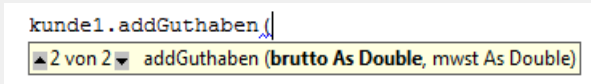
Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

BEISPIEL 3.21: Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

VB

```
Public Class CKunde
    ...
    Public Sub addGuthaben(betrag As Decimal)
        If _Stammkunde Then _Guthaben += betrag
    End Sub

    Public Sub addGuthaben(brutto As Double, mwst As Double)
        If _Stammkunde Then _Guthaben += CDec(brutto / (1 + mwst))
    End Sub
    ...
End Class
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.6.3 Statische Methoden

Genauso wie s*tatischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden werden ebenfalls mit dem *Shared*-Modifizierer gekennzeichnet und 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 3.22: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

VB
```
Public Class MeineFormeln

    Public Shared Function kreisUmfang(radius As Double) As Double
        Return 2 * Math.PI * radius
    End Function

    Public Shared Function kugelVolumen(radius As Double) As Double
        Return 4 / 3 * Math.PI * Math.Pow(radius, 3)
    End Function

    Public Shared Function Netto(brutto As Decimal, mwst As Decimal) As Decimal
        Return brutto / (1 + mwst)
    End Function
    ...

End Class
```

Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

HINWEIS: Leider kann bei Klassen die *With*-Anweisung nicht verwendet werden, da diese nur bei Objekten funktioniert.

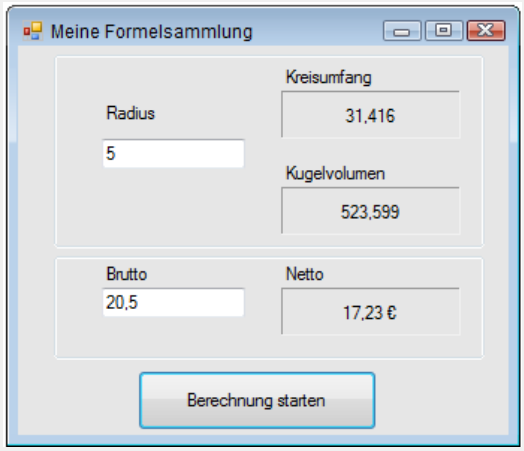
BEISPIEL 3.23: *(Fortsetzung)* Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
  Dim r As Double = Convert.ToDouble(TextBox1.Text)          ' Kreisradius konvertieren

  Label1.Text = MeineFormeln.kreisUmfang(r).ToString("0.000")
  Label2.Text = MeineFormeln.kugelVolumen(r).ToString("0.000")
```

BEISPIEL 3.23: ***(Fortsetzung)*** **Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.**

```
    Dim b As Double = Convert.ToDouble(TextBox2.Text)          ' Brutto konvertieren
    Label3.Text = MeineFormeln.Netto(b, 0.19).ToString("C")
End Sub
```

Ergebnis



3.7 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 .NET-Ereignismodell erhalten Sie im Kapitel 26 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.7.1 Ereignise deklarieren

Ein Ereignis fügen Sie der Klassendefinition über das *Event*-Schlüsselwort zu.

SYNTAX: `Public Event Ereignisname([Parameterdeklarationen])`

BEISPIEL 3.24: In der Klasse *CKunde* wird ein Ereignis mit dem Namen *GuthabenLeer* deklariert.

VB
```
Public Class CKunde
    Private _guthaben As Decimal

    Public Event GuthabenLeer(sender As Object, e As String)

    Public Sub New(betrag As Decimal)       ' Konstruktor
        ...
        _guthaben = betrag
    End Sub
    ...
```

3.7.2 Ereignis auslösen

Ausgelöst wird das Ereignis ebenfalls im Klassenkörper über das Schlüsselwort *RaiseEvent*.

SYNTAX: `RaiseEvent` *Ereignisname*([*Parameterwerte*])

BEISPIEL 3.25: *(Fortsetzung)* Das Ereignis *GuthabenLeer* "feuert" innerhalb der Methode *addGuthaben* dann, wenn das Guthaben den Wert von 10 € unterschreitet.

VB
```
    Public Sub addGuthaben(betrag As Decimal)
        _guthaben += betrag
        If _guthaben <= 10 Then
            Dim msg As String = "Das Guthaben beträgt nur noch" &
                                _guthaben.ToString("C") & "!"
            RaiseEvent GuthabenLeer(Me, msg)
        End If
    End Sub
    ...
End Class
```

Die Ereignisdefinition ist in diesem Beispiel bewusst einfach gehalten, um den Einsteiger nicht zu verschrecken. Normalerweise sollten Ereignisse immer zwei Parameter an die aufrufende Instanz übergeben: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse (siehe 20.2.2). Auf Letzteres haben wir der Einfachheit wegen verzichtet und stattdessen einen einfachen Meldungsstring übergeben.

3.7.3 Ereignis auswerten

Um dem Compiler mitzuteilen, dass das Objekt über Ereignisse verfügt, müssen Sie bei der Referenzierung der Objektvariablen vor dem Objektbezeichner das Schlüsselwort *WithEvents* einfügen.

SYNTAX: Private|Public `WithEvents` *Objektname* As *Klassenname*

BEISPIEL 3.26: ***(Fortsetzung)*** **Wir verwenden die im Vorgängerbeispiel definierte Klasse *CKunde* in einem Formular *Form1* mit einem *Button***

VB

```
Public Class Form1
```

Der Einfachheit wegen erledigen wir die Referenzierung und die Instanziierung der Objektvariablen *kunde1* in einem Schritt und weisen dabei dem Kunden ein Anfangsguthaben von 100 Euro zu.

```
Private WithEvents kunde1 As New CKunde(100)
```

Der Eventhandler für das Ereignis *GuthabenLeer* des Kunden:

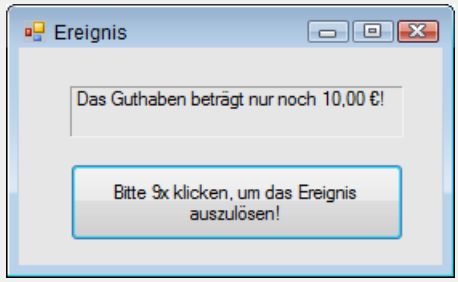
```
Private Sub kunde1_GuthabenLeer(sender As Object, e As String) Handles kunde1.GuthabenLeer
    Label1.Text = e.ToString
End Sub

End Class
```

Bei jedem Klick auf den *Button* wird das Guthaben des Kunden um 10 € verringert:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Kunde1.addGuthaben(-10)
End Sub
```

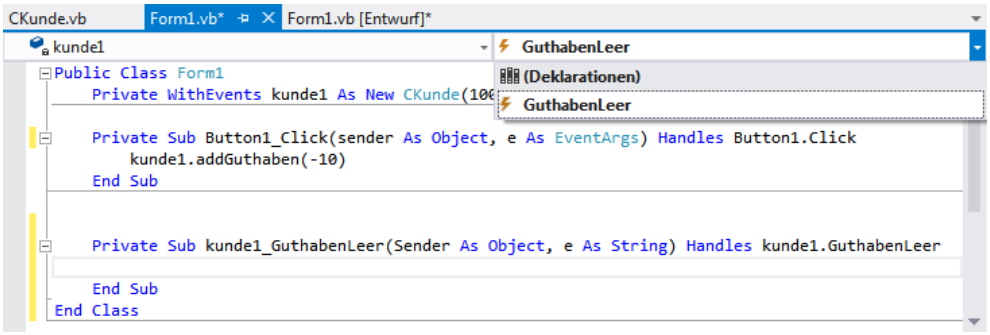
Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, feuert das Ereignis und im *Label* erscheint eine Meldung:



Visual Studio unterstützt Sie beim Erstellen des Rahmencodes der Event-Handler für selbst definierte Ereignisse natürlich genauso, wie Sie das z.B. für das *Click*-Ereignis eines *Button*s zur Genüge gewöhnt sind.

Nach dem Deklarieren der Objektvariablen wird in der linken oberen Klappbox das Objekt *kunde1* selektiert und in der rechten das Ereignis *GuthabenLeer*. Der Rahmencode des Eventhandlers wird automatisch generiert:



3.7.4 Benutzerdefinierte Ereignisse (Custom Events)

Meistens deklarieren Sie ein Ereignis mittels *Event*-Schlüsselwort unter Angabe eines Ereignisdelegaten, die Codegenerierung für die Ereignisverwaltung erfolgt automatisch.

BEISPIEL 3.27: Standardmäßige Ereignisdefinition in zwei Schritten

VB
```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Event NumberChanged As NumberChangedHandler
```

In der Regel ist diese Art der Deklaration auch völlig ausreichend, aber in einigen Fällen möchten Sie die Ereignisverwaltung doch lieber selbst in die Hand nehmen.

Benutzerdefinierte Ereignis-Accessoren (Custom Events) erlauben dem Programmierer die genaue Definition der Vorgänge, die beim Hinzufügen bzw. Entfernen eines Eventhandlers und beim Auslösen eines Events ablaufen sollen (siehe dazu auch die Ausführungen zum Microsoft Event Pattern im Kapitel 26).

Deklaration

Durch Deklarieren eines Custom Events wird dem Compiler mitgeteilt, dass er die Codegenerierung für die Ereignisverwaltung außer Kraft setzen soll, der Programmierer muss sich also nun selbst darum kümmern wie die Ereignishandler an-/abgemeldet und aufgerufen werden.

Ein Custom Event wird durch das der *Event*-Deklaration vorangestellte *Custom*-Schlüsselwort unter Angabe eines Ereignisdelegaten definiert.

BEISPIEL 3.28: Deklaration eines benutzerdefinierten Ereignisses *NumberChanged*

VB
```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Custom Event NumberChanged As NumberChangedHandler
```

Wenn Sie nach einer solchen Ereignisdeklaration die *Enter*-Taste drücken, erstellt Visual Studio automatisch den Rahmencode für die Ereignis-Acessoren (auf ähnliche Weise wie für Properties):

```
Public Custom Event NumberChanged As NumberChangedHandler
    AddHandler(value As NumberChangedHandler)

    End AddHandler

    RemoveHandler(value As NumberChangedHandler)

    End RemoveHandler

    RaiseEvent(i As Integer)

    End RaiseEvent
End Event
```

Wie Sie sehen, besteht die Deklaration eines Custom Events aus drei Sektionen, die unter folgenden Bedingungen abgearbeitet werden:

- *AddHandler*, wenn ein Ereignishandler entweder mittels *Handles*-Schlüsselwort oder *AddHandler*-Methode hinzugefügt wird.
- *RemoveHandler*, wenn ein Handler mittels *RemoveHandler*-Methode wieder entfernt wird.
- *RaiseEvent*, wenn ein Ereignis ausgelöst wird.

Sie müssen sich jetzt also selbst um das Hinzufügen und Entfernen der Handler zu bzw. aus ihrem Container (üblicherweise eine Collection), sowie um die Benachrichtigung der angeschlossenen Handler beim Auslösen des Ereignisses kümmern. Obwohl das mehr Codezeilen erfordert als eine standardmäßige Implementierung, haben Sie eine größere Flexibilität beim Programmieren.

Anwendung

Da das alles ziemlich verwirrend klingen mag, soll ein Beispiel Licht in die Dunkelheit bringen.

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

VB

Eine Klasse *CCounter* erzeugt im Sekundentakt eine Zahl von 1 bis 10 und löst dabei das benutzerdefinierte Ereignis *NumberChanged* aus. Als Container für die angemeldeten Eventhandler dient eine (generische) *List*:

```
Public Class CCounter

        Public Delegate Sub NumberChangedHandler(i As Integer)
        Private handlers As New List(Of NumberChangedHandler)

        Public Custom Event NumberChanged As NumberChangedHandler
            AddHandler(value As NumberChangedHandler)
                If handlers.Count <= 3 Then handlers.Add(value)
            End AddHandler
```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

```
            RemoveHandler(value As NumberChangedHandler)
                handlers.Remove(value)
            End RemoveHandler

            RaiseEvent(i As Integer)
                If i > 30 Then
                    For Each handler As NumberChangedHandler In handlers
                        handler.Invoke(i)
                    Next
                End If
            End RaiseEvent
        End Event
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
        Public Sub DoCount()
            For i As Integer = 1 To 10
                System.Threading.Thread.Sleep(1000)
                RaiseEvent NumberChanged(i * 10)
            Next
        End Sub
    End Class
```

Ein Eventhandler zeigt die erzeugten Zahlen in einer *ListBox* an:

```
    Private Sub c_NumberChanged(i As Integer)
        ListBox1.Items.Add(i.ToString)
    End Sub
```

Schließlich der Test der Klasse *CCounter*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Instanziierung:

```
        Dim c As New CCounter
```

Eventhandler anmelden:

```
        AddHandler c.NumberChanged, AddressOf c_NumberChanged
```

Methode aufrufen, die das Ereignis auslöst:

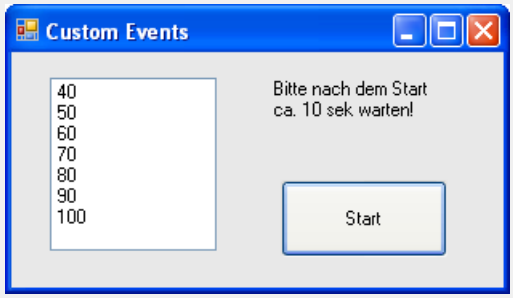
```
        c.DoCount()
```

Eventhandler abmelden:

```
        RemoveHandler c.NumberChanged, AddressOf c_NumberChanged
    End Sub
```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

Ergebnis



Wie Sie an diesem Beispiel sehen, ermöglichen Ihnen benutzerdefinierte Ereignisse eine freizügige Programmierung der Verwaltung Ihrer Ereignishandler. Im obigen Code haben wir als Container für die Ereignisdelegaten eine typisierte *List* gewählt, wir hätten aber auch z.B. eine (untypisierte) *ArrayList* nehmen können. In die *AddHandler*-Sektion wurde von uns eine Bedingung eingebaut, die die Anzahl der angeschlossenen Eventhandler auf 3 limitiert. Die *RaiseEvent*-Sektion bewirkt, dass die angeschlossenen Handler nur dann benachrichtigt werden, wenn die übergebene Zahl größer als 30 ist.

Ein nicht zu unterschätzender Vorteil von Custom Events ist auch das Vorhandensein eines zentralen Containers (im Beispiel die *handlers*-Liste), welcher alle angemeldeten Eventhandler innerhalb der Eventdeklaration kapselt.

Die doppelte Verwendung von *RaiseEvent* (innerhalb der *DoCount*-Methode und innerhalb der *Custom Event*-Ereignisdefinition) kann möglicherweise Verwirrung stiften. Deshalb sei hier nochmals auf den grundlegenden Unterschied hingewiesen: *RaiseEvent* in der *DoCount*-Methode löst das *NumberChanged*-Ereignis aus, die *RaiseEvent*-Sektion hingegen spezifiziert den beim Auslösen des Ereignisses auszuführenden "organisatorischen" Code, d.h., alle im Container (*List*) enthaltenen Eventhandler (Delegates) werden durchlaufen und aufgerufen.

3.8 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache, zu der Visual Basic ja mittlerweile auch gehört, realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

HINWEIS: Wenn Sie in einigen bisherigen Beispielen in den von Ihnen selbst entwickelten Klassen keinen eigenen Konstruktor definiert hatten, so wurde automatisch der von *System.Object* geerbte parameterlose *New*-Standardkonstruktor verwendet.

3.8.1 Der Konstruktor erzeugt das Objekt

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Konstruktor ist immer eine *Sub* mit dem Namen *New()*.

```
SYNTAX:  Public Sub New([Parameterliste])
             ' Parameter den Feldern der Klasse zuweisen
             ' Initialisierungscode ausführen
         End Sub
```

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

HINWEIS: 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, d h., der von *System.Object* geerbte parameterlose Konstruktor steht nicht mehr zur Verfügung!

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *Set*-Eigenschaftsprozedur mit dem Namen *New* hinzu. Als Parameter übergeben Sie die Werte für die Felder, die Sie initialisieren möchten.

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

VB

```
Public Class CKunde
```

Die (natürlich privaten) Zustandsvariablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
    Private _ort As String
    Private _stammkunde As Boolean
    Private _guthaben As Decimal
```

Der erste Konstruktor initialisiert nur zwei private Variablen:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

Der zweite Konstruktor initialisiert alle Variablen der Klasse. Um den Code etwas zu kürzen, wird für die ersten beiden Variablen der erste Konstruktor bemüht:

```
    Public Sub New(Anrede As String, Nachname As String, PLZ As Integer,
                   Ort As String, Stammkunde As Boolean, Guthaben As Decimal)
        Me.New(Anrede, Nachname)
```

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

```
        _plz = PLZ
        _ort = Ort
        _stammkunde = Stammkunde
        _guthaben = Guthaben
    End Sub

End Class
```

Aufruf

Da wir der Klasse *CKunde* zwei Konstruktoren hinzugefügt haben, ist die bisher gewohnte parameterlose Instanziierung von Objekten mit dem *New()*-Standardkonstruktor nicht mehr möglich (es sei denn, Sie fügen selbst eine weitere Überladung hinzu, die keine Parameter entgegen nimmt)!

BEISPIEL 3.31: (Fortsetzung) Zwei Objekte der oben deklarierten Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert. Für jedes Objekt wird ein anderer überladener Konstruktor verwendet.

VB
```
Try
      Dim kunde1 As New CKunde("Herr", "Müller")
      Dim kunde2 As New CKunde("Frau", "Hummel", 12345, "Berlin", True, 100)
      ' Dim kunde3 As New CKunde()             ' geht nicht mehr!!!
      MessageBox.Show("Objekte erfolgreich erzeugt!")
Catch ex As Exception
      MessageBox.Show("Fehler beim Erzeugen des Objekts!")
End Try
```

Sie sehen, dass das Initialisieren der Objekte viel einfacher geworden ist. Anstatt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt mit einer einzigen Anweisung.

3.8.2 Bequemer geht's mit einem Objekt-Initialisierer

Vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Kapitel 6) wurden so genannte Objekt-Initialisierer eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden.

Der Objekt-Initialisierer erwartet das Schlüsselwort *With* unmittelbar nach dem Erzeugen des Objekts. Anschließend folgt die in geschweiften Klammern eingeschlossene Liste der zu initialisierenden Mitglieder.

HINWEIS: Einem Objektinitialisierer können nur Eigenschaften oder öffentliche Felder übergeben werden, das Initialisieren privater Felder, wie im obigen Konstruktor-Beispiel, ist nicht möglich!

BEISPIEL 3.32: Erzeugen einer Instanz der Klasse *CPerson*

VB
```
Public Class CPerson
   Public Name As String
   Public Strasse As String
   Public PLZ As Integer
   Public Ort As String
End Class
```

Der Objektinitialisierer:

```
Dim person1 As New CPerson With {.Name = "Müller", .Strasse = "Am Waldesrand 7",
                                 .PLZ = 12345, .Ort = "Musterhausen"}
```

BEISPIEL 3.33: Verschachtelte Objektinitialisierung beim Erzeugen einer Instanz der Klasse *Rectangle*

VB
```
Dim rect As New Rectangle With {.Location = New Point With {.X = 3, .Y = 7},
                            .Size = New Size With {.Width = 19, .Height = 34} }
```

BEISPIEL 3.34: Initialisieren einer Collection aus Objekten

VB
```
Dim personen() = {New CPerson With {.Name = "Müller", .Strasse = "Am Wald 7",
                                    .PLZ = 12345, .Ort = "Musterhausen"},
                  New CPerson With {.Name = "Meier", .Strasse = "Hauptstr. 2",
                                    .PLZ = 2344, .Ort = "Walldorf"},
                  New CPerson With {.Name = "Schulz", .Strasse = "Wiesenweg 5",
                                    .PLZ = 32111, .Ort = "Biesdorf"}}
```

3.8.3 Destruktor und Garbage Collector räumen auf

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.

Der *Public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:
```
Protected Overrides Sub Finalize()
        ' hier Code für Aufräumarbeiten implementieren
End Sub
```

BEISPIEL 3.35: Destruktor und Garbage Collector

VB

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 Shared anzahl As Integer = 0
```

Konstruktor:

```
    Public Sub New()
        anzahl += 1
    End Sub
```

Quasi-Destruktor:

```
    Protected Overrides Sub Finalize()
        anzahl -= 1
        MyBase.Finalize()        ' Aufruf der Basisklassenmethode
    End Sub

End Class
```

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:

```
    Private kunde1 As CKunde            ' Objekt referenzieren
```

Objekt erzeugen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1 = New CKunde
    End Sub
```

Objekt entfernen:

```
    Private Sub Button2_Click(sender As Object, e AsEventArgs) Handles Button2.Click
        kunde1 = Nothing                 ' Objekt dereferenzieren
    End Sub
```

Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:

```
    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = CKunde.anzahl.ToString
    End Sub
```

BEISPIEL 3.35: Destruktor und Garbage Collector

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 ziemlich lange 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!

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit der *Finalize*-Methode ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode verwenden (siehe *Dispose*-Methode).

3.8.4 Mit Using den Lebenszyklus des Objekts kapseln

Auch mit dem Schlüsselwort *Using* kann man selbst für das sichere Erzeugen und Vernichten von Objekten sorgen. Voraussetzung dafür ist, dass das Objekt die *IDisposable*-Schnittstelle implemen-

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

tiert. Hinter den Kulissen wird ein *Try-Finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen.

BEISPIEL 3.36: Sicheres Erzeugen und Freigeben von ADO.NET-Objekten (siehe 23.3.5)

VB
```
...
Using conn As New SqlConnection(connString)
   Using cmd As New SqlCommand(cmdString, conn)
      conn.Open()
      cmd.ExecuteNonQuery()
   End Using
End Using
...
```

Ein weiteres Beispiel für *Using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File verwenden

3.9 Vererbung und Polymorphie

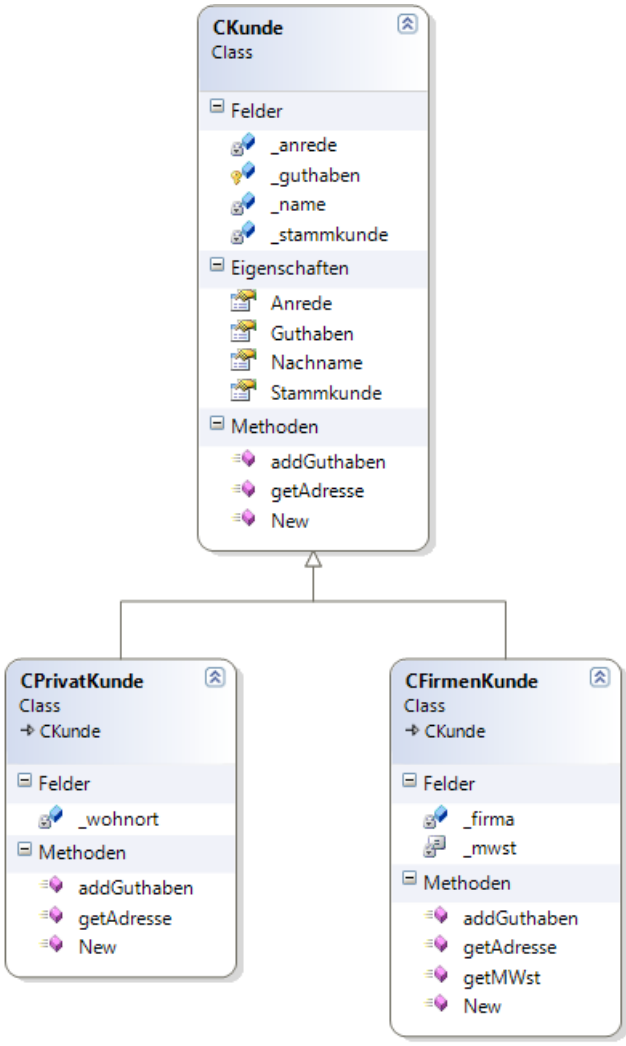
Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. Visual Basic 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!

3.9.1 Vererbungsbeziehungen im Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.



Das obige, mit Visual Studio erzeugte, Klassendiagramm 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 *New*-Methoden sind nichts weiter als die Konstruktoren der entsprechenden Klassen.

3.9.2 Überschreiben von Methoden (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* tauchen allerdings nochmals in den beiden Subklassen auf, warum? In unserem Beispiel handelt es sich um so genannte *überschriebene Methoden*, d h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

HINWEIS: Verwechseln Sie das *Überschreiben* von Methoden nicht mit dem in 2.8.5 beschriebenen *Überladen* von Methoden. Beides hat nichts, aber auch gar nichts, miteinander zu tun!

3.9.3 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 *Overridable*-Methodendeklarationen[1].

```
Public Class CKunde
```

Die privaten Felder:

```
    Private _anrede As String
    Private _name As String
    Private _stammkunde As Boolean
```

Auf das folgende Feld soll auch eine Subklasse zugreifen können:

```
    Protected _guthaben As Decimal
```

Der eigene Konstruktor:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

Die Eigenschaften:

```
    Public WriteOnly Property Anrede()
        Set(value)
```

[1] Eigentlich hätten wir die Klasse auch noch als *MustInherit* deklarieren müssen (siehe dazu 3.6.6).

```
            _anrede = value
        End Set
    End Property

    Public WriteOnly Property Nachname()
        Set(value)
            _name = value
        End Set
    End Property

    Public Property Stammkunde() As Boolean
        Get
            Return _stammkunde
        End Get
        Set(value As Boolean)
            _stammkunde = value
        End Set
    End Property

    Public ReadOnly Property Guthaben() As Decimal
        Get
            Return _guthaben
        End Get
    End Property
```

Die folgenden beiden Methoden sollen von den Subklassen überschrieben werden können:

```
    Public Overridable Function getAdresse() As String
        Return _anrede & " " & _name
    End Function

    Public Overridable Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
    End Sub

End Class
```

Subklassen

Die erste Methode in der Subklasse ist in der Regel der Konstruktor. Dieser Konstruktor benutzt das *MyBase*-Schlüsselwort, um den Konstruktor der Basisklasse aufzurufen. Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Das *Overrides*-Schlüsselwort der beiden Funktionen bedeutet, dass hier die in der Basisklasse definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

HINWEIS: Wenn Sie das *Overrides*-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.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde
    Private _wohnort As String
```

Der Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)  ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methoden werden überschrieben:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _wohnort
    End Function

    Public Overrides Sub addGuthaben(geld As Decimal)
```

5% des Rechnungsbetrags werden als Guthaben angerechnet (Direktzugriff auf die *Protected*-Variable der Basisklasse *CKunde*):

```
        _guthaben += 0.05 * geld
     End Sub
End Class
```

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in einigen Details:

```
Public Class CFirmenKunde
    Inherits CKunde
    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Konstruktor (notwendig, weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _firma
    End Function
```

```
    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))         ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01D)        ' 1% als Guthaben angerechnet
    End Sub
```

Eine normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function

End Class
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *Overrides*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *Overridable* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde                          ' erbt von der Basisklasse CKunde!
    Private _wohnort As String
```

Ein eigener Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)                 ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _wohnort
    End Function
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    Public Overrides Sub addGuthaben(geld As Decimal)
```

Hier erfolgt ein Direktzugriff auf die *Protected*-Variable *_guthaben* der Basisklasse *CKunde*:

```
        _guthaben += 0.05 * geld
    End Sub

End Class
```

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
    Inherits CKunde

    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Auch hier ist ein Konstruktor notwendig (weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _firma
    End Function

    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))  ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01)                   ' 1% als Guthaben angerechnet
    End Sub
```

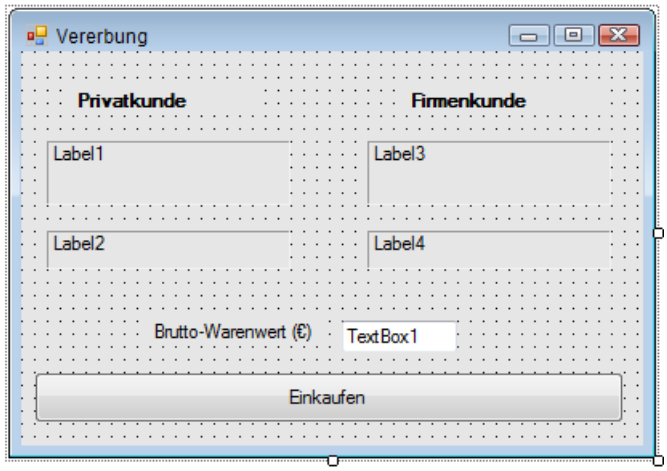
Und zum Schluss noch eine ganz normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function
End Class
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie folgende Benutzerschnittstelle:



3.9.4 Objekte implementieren

Es genügt, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten:

```
Public Class Form1
    Private kunde1 As CPrivatKunde
    Private kunde2 As CFirmenKunde
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Fügen Sie zunächst den Rahmencode des Konstruktors hinzu, indem Sie in den beiden Comboboxen am oberen Rand des Codefenster Folgendes einstellen: *Klassenname = Form1*, *Methodenname = New.*

Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

```
Public Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()
        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.

        kunde1 = New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False

        kunde2 = New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        TextBox1.Text = "100"
    End Sub
```

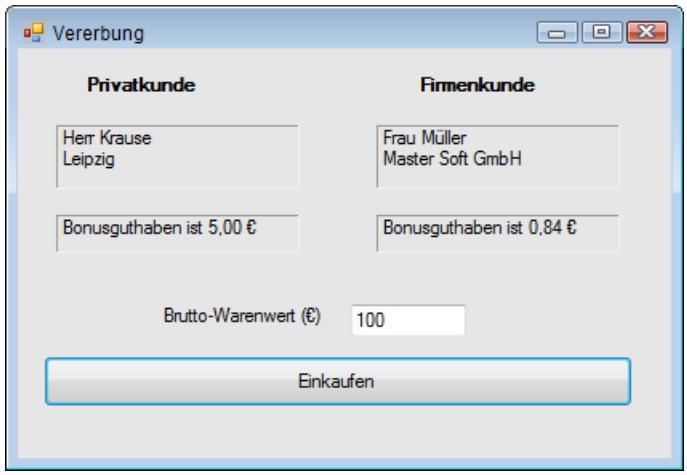
Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = 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")
    End Sub
End Class
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.



Die Werte in der 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.

3.9.5 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 3.37: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

VB

```
Public Class CKunde                    ' Basisklasse
    ...
    Public Sub test()
        MessageBox.Show("Hallo Kunde!")
    End Sub
End Class
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatkunde* hinzu:

```
Public Class CPrivatkunde              ' abgeleitete Klasse
    Inherits CKunde

    Public Sub test()
        MessageBox.Show("Hallo Privatkunde!")
    End Sub

End Class
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *Sub "test" führt Shadowing für einen überladbaren Member durch, der in der Basisklasse CKunde deklariert ist. Wenn Sie die Basismethode überladen möchten, muss die Methode als "Overloads" deklariert werden.*

Sie lassen diese Warnung unbeachtet. Der Test erfolgt in *Form1*:

```
Public Class Form1
 Private kunde1 As New CPrivatkunde()

 Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
     kunde1.test()
 End Sub
End Class
```

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 Schlüsselwort *Overloads* markieren.

BEISPIEL 3.38: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

VB
```
Public Class CPrivatkunde            ' abgeleitete Klasse
  Inherits CKunde
  ...
  Public Overloads Sub test()
      MessageBox.Show("Hallo Privatkunde!")
  End Sub

End Class
```

3.9.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennen gelernt 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 *Overridable* deklariert werden.
- Fehlt das Schlüsselwort *Overridable* 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 *Overrides* deklariert werden.
- Wenn Sie das *Overrides*-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 das der Klassendeklaration nachgestellte Schlüsselwort *Inherits* bekannt gemacht:

SYNTAX:
```
Public Class SubKlasse
Inherits Basisklasse
     ' ... Implementierungscode
End Class
```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *MyBase*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *Me*-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 (*MyBase*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

3.9.7 Polymorphe Methoden

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 Kompilierens 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 3.39: Polymorphe Methoden

VB

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, die diesmal in einem Array mit drei Feldern vom Typ der Basisklasse *CKunde* abgelegt werden sollen:

```
Public Class Form1
    Private Kunden(2) As CKunde          ' Referenz auf Basisklasse CKunde!
```

Im Konstruktorcode des Formulars erzeugen wir einen Privat- und zwei Firmenkunden:

```
    Public Sub New()
        InitializeComponent()
        Dim kunde1 As New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False
        Dim kunde2 As New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        Dim Kunde3 As New CFirmenKunde("Herr", "Maus", "Manfreds Internet AG")
        Kunde3.Stammkunde = False
```

Unser eingangs deklariertes Array nimmt nun Privat- und Firmenkunden in wahlloser Reihenfolge auf:

BEISPIEL 3.39: Polymorphe Methoden

```
        Kunden(0) = kunde1
        Kunden(1) = kunde2
        Kunden(2) = Kunde3
        TextBox1.Text = "100"
    End Sub
```

Das Array wird zunächst in einer *For-Next*-Schleife durchlaufen. Dabei werden die polymorphen Methoden (das sind die mit *Overridable* bzw. *Overrides* deklarierten) für alle Objekte aufgerufen. Es werden also sowohl die Adressen ausgegeben als auch für jeden Kunden das Guthaben berechnet, nachdem er für 100 € Waren gekauft hat.

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
        For i As Integer = 0 To Kunden.Length - 1
            Kunden(i).addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & Kunden(i).getAdresse & vbCrLf &
                          Kunden(i).Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sind, "weiß" das Programm zur Laufzeit genau, welche Implementierung der Methoden *getAdresse* und *addGuthaben* für den Privat- und für den Firmenkunden die richtige ist: Genau darin liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.40: Die alternative Implementierung obigen Codes mittels *For Each*-Schleife bringt das Problem der Polymorphie noch deutlicher auf den Punkt

VB

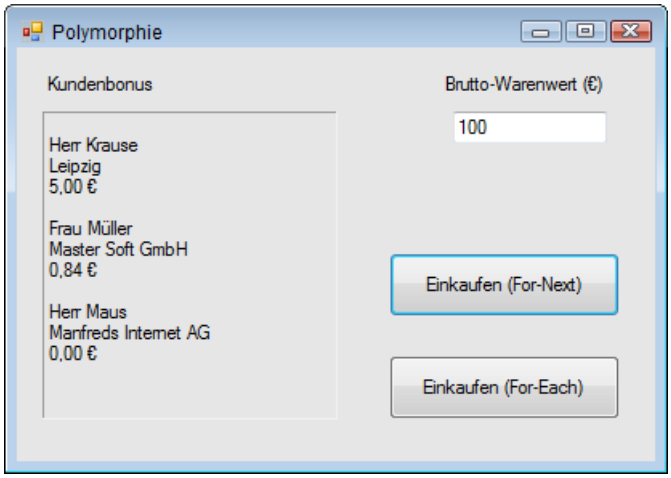
```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
```

Die Schleifenvariable *ku* ist eine Referenz auf die Basisklasse *CKunde*:

```
        For Each ku As CKunde In Kunden
            ku.addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & ku.getAdresse & vbCrLf &
                                        ku.Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
End Class
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verknüpft sind. Egal ob Privat- oder Firmenkunde – es werden immer die passenden Methodenimplementierungen aufgerufen:



HINWEIS: 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.

3.10 Besondere Klassen und Features

Zum Schluss wollen wir noch auf einige wichtige Klassen und Features eingehen, die in der OOP eine besondere Rolle spielen.

3.10.1 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, können diese mit dem Schlüsselwort *MustInherit* gekennzeichnet werden.

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

VB
```
Public MustInherit Class CKunde
...
End Class
```

Während die Referenzierung nach wie vor möglich ist

```
Dim Kunde As CKunde
```

[1] Können Sie sich vielleicht vorstellen, wie eine Instanz der Klasse *Fahrzeug* konkret aussehen soll?

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

schlägt der Versuch einer Instanziierung fehl:

```
Kunde = New CKunde("Herr", "Krause")              ' Fehler
```

HINWEIS: Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, der Schnittstelle (siehe Abschnitt 3.10.1).

3.10.2 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 *Overrides* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *MustOverride*.

HINWEIS: Die Deklaration einer abstrakten Methode erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.42: Die Funktion *getAdresse* einer abstrakten *CKunde*-Klasse wird in den Subklassen komplett überschrieben und kann deshalb anstatt mit *Overridable* mit *MustOverride* deklariert werden.

VB
```
Public MustInherit Class CKunde
...
  Public MustOverride Function getAdresse() As String       ' abstrakte virtuelle Methode
...
End Class

Public Class CPrivatKunde
    Inherits CKunde
...
    Public Overrides Function getAdresse() As String  ' überschreibt Methode der Basisklasse
        Return _Anrede & "  " & _Name & "  " & _Wohnort
    End Function
...
End Class
```

3.10.3 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 *NotInheritable* schützen.

BEISPIEL 3.43: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

VB
```
Public NotInheritable Class CPrivatKunde
    Inherits CKunde
    ...
End Class
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
Public Class CStudent           ' Fehler!!!
    Inherits CPrivatKunde
    ...
End Class
```

HINWEIS: Eine versiegelte Klasse kann niemals eine Basisklasse sein. Vererbungsmodifikatoren wie *MustInherit* und *Overridable* 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.

3.10.4 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 (z.B. *Form1.Designer.vb*) vom Code des Entwicklers (*Form1.vb*) getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms-Projekts selbst überzeugen kann).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *Partial* zu kennzeichnen.

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

VB
```
Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0

        Public Property NachName() As String
            Get
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property
```

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

```
        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
    End Class
```

Obige Klasse könnte (als eine von mehreren Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
Partial Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0
End Class

Public Class CKunde
        Public Property NachName() As String
            Get
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property

        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
End Class

Partial Public Class CKunde
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
End Class
```

Wie Sie sehen, kann (muss aber nicht) bei einer der partiellen Definitionen auf *Partial* verzichtet werden, diese Klasse ist dann gewissermaßen die Ausgangsklasse, die durch den Code der anderen partiellen Klassen erweitert wird.

HINWEIS: Auch *Structure*- oder *Interface*-Definitionen können mittels *Partial*-Modifizierer gesplittet werden!

3.10.5 Die Basisklasse System.Object

Jedes Objekt in .NET ist von der Basisklasse *System.Object* abgeleitet. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

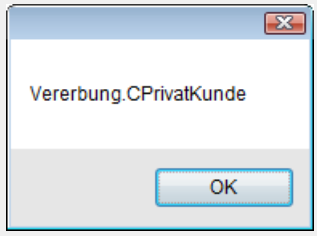
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird 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 3.45: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier Vererbung) und die Klassenzugehörigkeit von *kunde1*.

VB
```
MessageBox.Show(kunde1.ToString)
```

Ergebnis



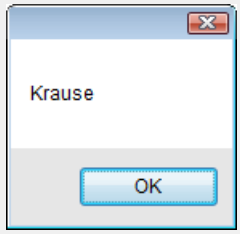
Sie können das standardmäßige Verhalten von *ToString* mittels *Overrides*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

VB
```
Public Overrides Function ToString() As String
   Return _name
End Function
```

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

Ergebnis



3.10.6 Property-Accessors

Die *Get*- und *Set*- Accessoren von Eigenschaften hatten in den ersten .NET-Versionen von Visual Basic die gleiche Sichtbarkeit wie die Eigenschaft zu der sie gehören. Seit .NET 2.0 ist es möglich, den Zugriff auf einen dieser Accessoren zu beschränken. Meist ist dies für den *Set*-Accessor sinnvoll, während der *Get*-Accessor in der Regel öffentlich bleiben soll.

BEISPIEL 3.47: Eine Eigenschaft mit *Get*- und *Set*-Accessoren. Der *Get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *Set*-Accessor nur einen *Friend*-Zugriff erlaubt.

VB

```
Private _knr As String
Public Property KontoNummer() As String
    Get
        Return _knr
    End Get
    Friend Set(value As String)
        _knr = value
    End Set
End Property
```

3.10.7 Nullbedingter Operator

Dieser mit VB 2015 eingeführte Operator ist eine bequeme Abkürzung für all jene Fälle, in denen auf *Nothing* zu prüfen ist:

BEISPIEL 3.48: Nullbedingter Operator

VB

```
Dim frm = firma.Addresse?.Land
```

ist eine Abkürzung für

```
Dim _tmp = firma.Addresse
Dim frm = If(_tmp IsNot Nothing, _tmp.Addresse.Land, Nothing)
```

BEISPIEL 3.49: Es wird nur dann abgezweigt, wenn der Bewerber zugewiesen und älter als 60 Jahre ist

VB
```
...
If bewerber?.Age > 60 Then ...
...
```

BEISPIEL 3.50: Ein Standardnamen wird zugeteilt wenn der Kunde nicht zugewiesen ist

VB
```
...
Dim name = If(kunde?.Name, "Müller")
...
```

BEISPIEL 3.51: Nur wenn ein *kunde*-Objekt vorhanden ist wird *ToString*() ausgeführt

VB
```
...
Dim s = kunde?.ToString()
...
```

3.11 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.

Einen Ausweg bietet die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile.

HINWEIS: Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.10.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

3.11.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Anstatt des Schlüsselworts *Class* verwenden Sie aber *Interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle immer mit "I" beginnen.

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

BEISPIEL 3.52: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert [2]

VB
```
Public Interface IPerson
  Property Nachname As String
  Property Vorname As String
  Function getName() As String
End Interface
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (***Public*** *Property* ...), diese aber haben 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!

3.11.2 Implementieren einer Schnittstelle

Die implementierende Klasse benutzt anstatt des Schlüsselworts *Inherits* das Schlüsselwort *Implements*.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.53: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

VB
```
Class CKunde
   Implements IPerson
        ...
```

Die von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
        Public Property Nachname As String Implements IPerson.Nachname
        Public Property Vorname As String Implements IPerson.Vorname

        Public Function getName() As String Implements IPerson.getName
           Return _Vorname & " " & _Nachname
        End Function
```

Es folgen die normalen Klassenmitglieder:

```
        ...
End Class
```

Den kompletten Code finden Sie im Praxisbeispiel

► 3.12.4 Schnittstellenvererbung verstehen

[2] Das Interface verwendet selbstimplementierende Eigenschaften.

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

3.11.3 Abfragen, ob eine Schnittstelle vorhanden ist

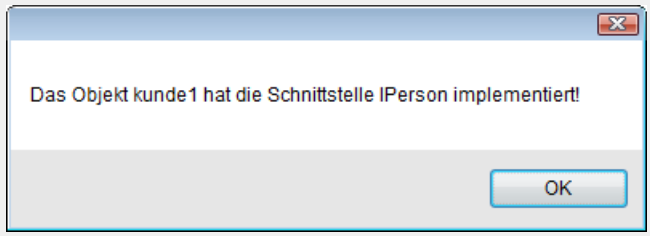
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *TypeOf*-Operator.

BEISPIEL 3.54: Wir ergänzen die Bedienoberfläche des Vorgängerbeispiels um eine zweite Schaltfläche.

VB

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If TypeOf kunde1 Is IPerson Then
        MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!")
    End If
End Sub
```

Ergebnis



3.11.4 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 3.55: Eine Klasse implementiert zwei Schnittstellen

VB

```
Public Class CPrivatkunde
    Implements IPerson, IKunde
    ...
End Class
```

3.11.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Anstatt 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).

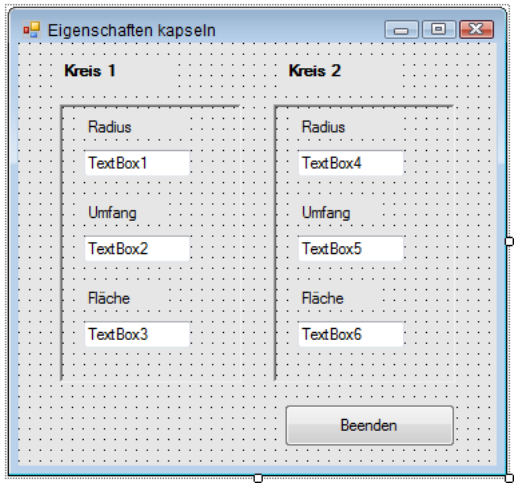
3.12 Praxisbeispiele

3.12.1 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 Beispiel handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *Radius*, *Umfang* und *Fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *radZ*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen weiteren Vorteil der OOP zu demonstrieren, d.h. ohne viel Mehraufwand beliebig viele Instanzen aus einer Klasse bilden, 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.vb*.

```
Public Class CKreis
```

Die private Zustandsvariable speichert den Wert des Radius:

```
    Private radZ As Double
```

Die Eigenschaft *Radius* (Rahmencode wird automatisch erzeugt!):

```
    Public Property Radius() As String
        Get
            Return radZ.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value)
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * radZ).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value) / 2 / Math.PI
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(radZ, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                radZ = 0
```

```
            End If
        End Set
    End Property
End Class
```

Wechseln Sie nun in den Klassencode von *Form1*.

Quellcode Form1

```
Public Class Form1
```

Ein Objekt wird erzeugt:

```
    Private Kreis1 As New CKreis()
```

Die folgenden Event-Handler sind einfach und übersichtlich, da die Objekte die inneren Funktionalitäten wegkapseln. Den Radius ändern:

```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        With Kreis1
            .Radius = TextBox1.Text
            TextBox2.Text = .Umfang
            TextBox3.Text = .Fläche
        End With
    End Sub
```

Den Umfang ändern:

```
    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        With Kreis1
            .Umfang = TextBox2.Text
            TextBox1.Text = .Radius
            TextBox3.Text = .Fläche
        End With
    End Sub
```

Die Fläche ändern:

```
    Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        With Kreis1
            .Fläche = TextBox3.Text
            TextBox1.Text = .Radius
            TextBox2.Text = .Umfang
        End With
    End Sub
```

Der Code für *Kreis2* ist analog aufgebaut und braucht deshalb hier nicht wiederholt zu werden (siehe Beispieldaten).

```
End Class
```

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. Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Objektinitialisierer

Man kann ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen Konstruktor gibt. Sie könnten also im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften wie folgt vornehmen:

```
Private Kreis1 As New CKreis With {.Radius = "1.0"}
```

HINWEIS: Mehr zu Objektinitialisierern erfahren Sie im Abschnitt 3.8.2!

3.12.2 Eine statische Klasse anwenden

Als "statisch" wollen wir hier solche Klassen bezeichnen, die lediglich statische (*Shared*) Mitglieder haben. Solche Klassen eignen sich beispielsweise ideal für Formelsammlungen (siehe *Math*-Klasse), da keine Objekte erzeugt werden müssen, denn es kann gleich "losgerechnet" werden. Das vorliegende Beispiel demonstriert dies anhand einer statischen Klasse *CKugel* zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt).

$V = 4/3 * Pi * r^3$

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 *Form*ular mit zwei *TextBox*en zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitansicht).

Quellcode CKugel

Statische Funktionen werden mit dem Schlüsselwort *Shared* gekennzeichnet.

```
Public Class CKugel
    Public Shared Function Durchmesser_Volumen(durchmesser As String) As Double
        Dim dur As Double = System.Double.Parse(durchmesser)
        Dim vol As Double = Math.Pow(dur, 3) * Math.PI / 6.0
```

```
        Return vol
    End Function

    Public Shared Function Volumen_Durchmesser(volumen As String) As Double
        Dim vol As Double = System.Double.Parse(volumen)
        Dim dur As Double = Math.Pow(6 / Math.PI * vol, 1 / 3.0)
        Return dur
    End Function
End Class
```

Quellcode Form1

Die Verwendung der Klasse im Formularcode:

```
Public Class Form1
```

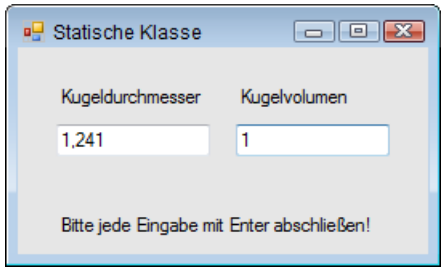
Die Berechnung startet nach Betätigen der *Enter*-Taste:

```
 Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter And TextBox1.Text <> String.Empty Then
            TextBox2.Text = CKugel.Durchmesser_Volumen(TextBox1.Text).ToString("#,##0.000")
        End If
    End Sub

    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        If e.KeyCode = Keys.Enter And TextBox2.Text <> String.Empty Then
            TextBox1.Text = CKugel.Volumen_Durchmesser(TextBox2.Text).ToString("#,##0.000")
        End If
    End Sub
End Class
```

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:



3.12.3 Vom fetten zum dünnen 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 damit die gesamte 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

Siehe Laufzeitansicht.

Quellcode (Fat Client)

```
Public Class Form1
```

Über die Bedeutung der folgenden drei globalen Zustandsvariablen brauchen wir wohl keine weiteren Worte zu verlieren:

```
    Private op As Char                  ' aktueller Operator   (+, - , *, /)
    Private reg1 As String = Nothing    ' erstes Register   (Operand)
    Private reg2 As String = Nothing    ' zweites Register  (Operand)
```

Wir wollen zur Steuerung des Programmablaufs eine spezielle Variable *state* verwenden, die den aktuellen Zustand speichert:

```
Private state As Byte = 1           ' aktuelles Register (1 oder 2)
```

Typisch für die nun folgenden Ereignisbehandlungen ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind.

Zur Eingabe einer Ziffer (0...9) benutzt der gesamte Ziffernblock eine gemeinsame Ereignisbehandlung:

```
Private Sub ButtonZ_Click(sender As Object, e As EventArgs)  Handles Button1.Click,
     Button2.Click, Button3.Click, Button4.Click, Button5.Click, Button6.Click,
        Button7.Click, Button8.Click, Button9.Click, Button10.Click, Button11.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1                ' zum ersten Operanden hinzufügen:
            reg1 &= cmd.Text.Chars(0)
            Label1.Text = reg1
        Case 2                ' zum zweiten Operanden hinzufügen:
            reg2 &= cmd.Text.Chars(0)
            Label1.Text = reg1 & " " & op & " " & reg2
    End Select
End Sub
```

Für die Eingabe der Operation (+, -, *, /) wird ähnlich verfahren:

```
Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
              Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1
            op = cmd.Text.Chars(0)      ' neuer Operand ...
            state = 2                   ' ... und Zustandswechsel
        Case 2
            ergebnis()        ' erst Zwischenergebnis mit altem Operand ermitteln ...
            op = cmd.Text.Chars(0)               ' ... dann neuer Operand
     End Select
     Label1.Text = reg1.ToString & " " & op
     reg2 = Nothing                              ' Reg2 löschen
 End Sub
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis in *reg1*:

```
 Private Sub ergebnis()
    Dim r1 As Double = Convert.ToDouble(reg1)
    Dim r2 As Double = Convert.ToDouble(reg2)
    Select Case op
        Case "+"c
            reg1 = (r1 + r2).ToString
        Case "-"c
            reg1 = (r1 - r2).ToString
```

```
            Case "*"c
                reg1 = (r1 * r2).ToString
            Case "/"c
                reg1 = (r1 / r2).ToString
        End Select
        reg2 = Nothing  ' löscht zweites Register
    End Sub
```

Die Ergebnistaste (=):

```
    Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
        If state = 2 Then
            ergebnis()
            Label1.Text &= " = " & reg1
            state = 1
        Else
            Label1.Text = reg1
            reg2 = Nothing   ' löscht zweites Register
        End If
    End Sub
```

Letztes eingegebenes Zeichen löschen (CE):

```
    Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
        Select Case state
            Case 1
                If Not (reg1 = Nothing) Then
                    reg1 = reg1.Remove(reg1.Length - 1, 1)
                    Label1.Text = reg1
                End If
            Case 2
                If Not (reg2 = Nothing) Then
                    reg2 = reg2.Remove(reg2.Length - 1, 1)
                    Label1.Text = reg2
                End If
        End Select
    End Sub
```

Alle Register sowie Anzeige löschen und Anfangszustand herstellen:

```
    Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
        reg1 = Nothing : reg2 = Nothing : Label1.Text = String.Empty
        state = 1
    End Sub
```

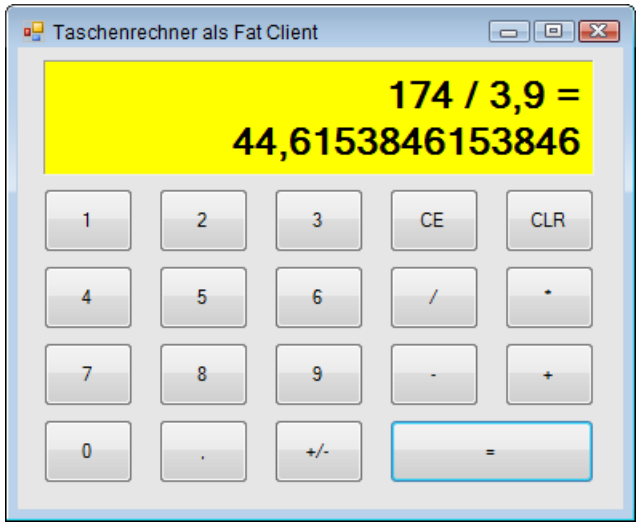
Schließlich noch der Vorzeichenwechsel (+/-):

```
    Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
        Dim r As Double
        Select Case state
            Case 1
                r = -Convert.ToDouble(reg1)
```

```
                reg1 = r.ToString
                Label1.Text = reg1
            Case 2
                r = -Convert.ToDouble(reg2)
                reg2 = r.ToString
                Label1.Text = reg1 & " " & op & " " & reg2
        End Select
    End Sub
End Class
```

Test

Der Vorzug gegenüber üblichen Rechnern (oder auch dem im Windows-Zubehör) sticht sofort ins Auge: Man kann den Rechenvorgang mitverfolgen, weil der komplette Ausdruck angezeigt wird.



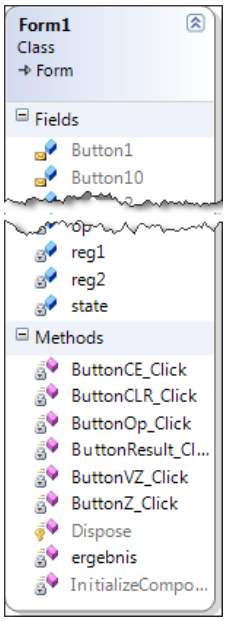
HINWEIS: Wie bei jedem anderen einfachen Taschenrechner auch, bleibt hier die Rangfolge der Operatoren (Punktrechnung geht vor Strichrechnung) unberücksichtigt. Bei der Eingabe von mehreren Operationen hintereinander, z.B. 3 + 4 * 12, ist deshalb zu beachten, dass erst die höherwertige Operation auszuführen ist (4 * 12).

Bemerkungen zum fetten Client

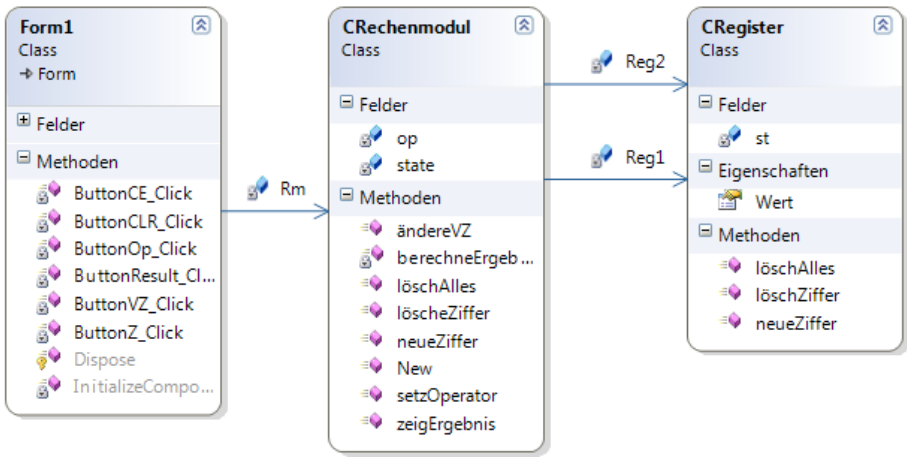
- Den Programmablauf könnte man in Gestalt eines Zustandsüberführungsdiagramms (*State Chart*) noch anschaulicher darstellen (siehe Kapitel 25).
- Leider ist die gesamte Intelligenz der Anwendung in der Benutzerschnittstelle *Form1* enthalten, also ein typischer "fetter" Client. Transparenz, Wiederverwendbarkeit und Wartbarkeit des Codes sind demzufolge katastrophal! Wie man das Programm auf ein höheres objektorientiertes Niveau heben kann, soll die folgende alternative Realisierung unseres Taschenrechners zeigen.

Abmagerungskur für den fetten Client

Dass es sich bei unserem alten Programm tatsächlich um einen Fat Client handelt, zeigt das zugehörige Klassendiagramm. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf *Form1.vb* 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:



Die Klasse *Form1* (Thin Client) ist gegenüber dem Vorgänger deutlich abgemagert und beschränkt sich nur noch auf ihre eigentliche Aufgabe, nämlich die Verwaltung der Benutzerschnittstelle. Die Klassen *CRechenmodul* und *CRegister* stellen hingegen die zweischichtige Geschäftslogik (Server) der Anwendung dar, kapseln also die Intelligenz der Anwendung.

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").

Quellcode für CRegister

```
Public Class CRegister
```

Die globale Variable *st* speichert den Registerinhalt als Zeichenkette:

```
    Private st As String = Nothing
```

Zugriff auf den numerischen Wert von *st:*

```
    Public Property Wert() As Double
        Get
            Try
                Wert = Convert.ToDouble(st)
            Catch
                Wert = 0
            End Try
        End Get
        Set(value As Double)
            st = value.ToString
        End Set
    End Property
```

Hinzufügen einer einzelnen Ziffer und Rückgabe des Anzeigestrings:

```
    Public Function neueZiffer(z As Char) As String
        If (Char.IsDigit(z) Or z = ",") Then
            st &= z
```

```
            Return (st)
        Else
            Return (String.Empty)
        End If
    End Function
```

Letzte Ziffer löschen und Anzeigestring zurückgeben:

```
    Public Function löschZiffer() As String
        If st.Length > 0 Then
            st = st.Remove(st.Length - 1, 1)
        End If
        Return (st)
    End Function
```

Gesamtes Register löschen:

```
    Public Sub löschAlles()
        st = String.Empty
    End Sub

End Class
```

Quellcode für CRechenmodul

```
Public Class CRechenmodul

    Private state As Byte = 1          ' Startmodus (Zustandsvariable)
    Private op As Char                 ' aktueller Operator
    Private Reg1, Reg2 As CRegister    ' zwei Rechenregister
```

Im Konstruktor werden zwei Register-Objekte erzeugt:

```
    Public Sub New()
        Reg1 = New CRegister()
        Reg2 = New CRegister()
    End Sub
```

Zifferneingabe in aktuelles Register:

```
    Public Function neueZiffer(z As Char) As String
        If state = 1 Then                       ' zum ersten Register hinzufügen:
            Return (Reg1.neueZiffer(z))
        Else                                    ' zum zweiten Register hinzufügen:
            Return (Reg1.Wert.ToString & " " & op & " " & Reg2.neueZiffer(z))
        End If
    End Function
```

Letzte Ziffer des aktuellen Registers löschen und resultierenden Registerinhalt zurückgeben:

```
    Public Function löscheZiffer() As String
        If state = 1 Then
            Return (Reg1.löschZiffer())
```

```
        Else
            Return (Reg2.löschZiffer())
        End If
    End Function
```

Vorzeichen des aktuellen Registers umkehren:

```
    Public Function ändereVZ() As String
        If state = 1 Then
            Reg1.Wert = -Reg1.Wert
            Return (Reg1.Wert.ToString)
        Else
            Reg2.Wert = -Reg2.Wert
            Return (Reg1.Wert.ToString & " " & op & " " & Reg2.Wert.ToString)
        End If
    End Function
```

Der Operator wird übernommen. Rückgabewert ist der String des ersten Operanden mit abschließendem Operatorenzeichen:

```
    Public Function setzOperator(o As Char) As String
        If state = 1 Then
            state = 2
        Else
            berechneErgebnis()               ' Zwischenergebnis (mit altem Operator) ermitteln
        End If
        op = o                                          ' neuen Operator übernehmen
        Reg2.löschAlles()                               ' zweites Register löschen
        Return (Reg1.Wert.ToString & " " & op)
    End Function
```

Die abschließende Rechenoperation ausführen und das Ergebnis liefern:

```
    Public Function zeigErgebnis() As String
        Dim s As String = ""
        If state = 1 Then      ' im Startmodus wird noch nichts berechnet, ...
            Reg2.löschAlles()    ' ... lediglich zweites Register gelöscht
        Else
            s = " = " & berechneErgebnis()
            state = 1
        End If
        Return (s)
    End Function
```

Eine Hilfsmethode zum Ausführen der Rechenoperation nebst Abspeichern des Ergebnisses im ersten Register (überschreibt erstes Register mit Ergebnis der Operation):

```
    Private Function berechneErgebnis() As String
        Select Case op
            Case "+"c
                Reg1.Wert = Reg1.Wert + Reg2.Wert
            Case "-"c
```

```
                Reg1.Wert = Reg1.Wert - Reg2.Wert
            Case "*"c
                Reg1.Wert = Reg1.Wert * Reg2.Wert
            Case "/"c
                Reg1.Wert = Reg1.Wert / Reg2.Wert
        End Select
        Reg2.löschAlles()       ' zweites Register löschen
        Return Reg1.Wert.ToString
    End Function
```

Alle Register löschen und Startzustand wieder herstellen:

```
    Public Sub löschAlles()
        Reg1.löschAlles()
        Reg2.löschAlles()
        state = 1
    End Sub
End Class
```

Quellcode für Form1

Die Oberfläche unseres Thin Client entspricht 100%ig der seines "fetten" Vorgängers. Die Programmierung ist allerdings – dank des *CRechenmodul*-Objekts – deutlich einfacher und transparenter geworden:

```
Public Class Form1
```

Einzige globale Variable ist eine Instanz der Klasse *CRechenmodul*:

```
    Private Rm As New CRechenmodul()
```

Eine Ziffer eingeben (0..9):

```
     Private Sub ButtonZ_Click(sender As Object, e As EventArgs) Handles Button1.Click,
           Button9.Click, Button8.Click, Button7.Click, Button6.Click, Button5.Click,
              Button4.Click, Button3.Click, Button2.Click, ButtonKomma.Click, Button0.Click
        Dim cmd As Button = CType(sender, Button)
        Label1.Text = Rm.neueZiffer(cmd.Text.Chars(0))
    End Sub
```

Die Operation eingeben (+, -, *, /):

```
     Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
           Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
        Dim cmd As Button = CType(sender, Button)
        Label1.Text = Rm.setzOperator(cmd.Text.Chars(0))
    End Sub
```

Ergebnis anzeigen (=):

```
    Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
        Label1.Text &= Rm.zeigErgebnis
    End Sub
```

Letztes eingegebenes Zeichen löschen (CE):

```
    Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
        Label1.Text = Rm.lösche Ziffer()
    End Sub
```

Alle Register sowie Anzeige löschen und Anfangszustand wieder herstellen:

```
    Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
        Rm.löschAlles()
        Label1.Text = String.Empty
    End Sub
```

Vorzeichenwechsel (+/-):

```
    Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
        Label1.Text = Rm.ändereVZ()
    End Sub

End Class
```

Test

Sie werden keinerlei Unterschied in Aussehen und Funktion 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 im Praxisbeispiel 3.12.8.

3.12.4 Schnittstellenvererbung verstehen

Ein mächtiges Feature der OOP ist das Konzept der Schnittstellenvererbung (siehe Abschnitt 3.11). Die Schnittstellenvererbung erschließt sich dem Einsteiger am leichtesten, wenn er sich vorher das Konzept abstrakter Klassen und Methoden (siehe Abschnitt 3.10.1) verinnerlicht hat.

In unserem Demobeispiel wollen wir Geldbeträge auf das Konto eines Kunden einzahlen bzw. von dort abheben. Die erste Lösung soll mittels einer abstrakten Klasse erfolgen[1]. Die zweite Lösung benutzt eine Schnittstelle (Interface).

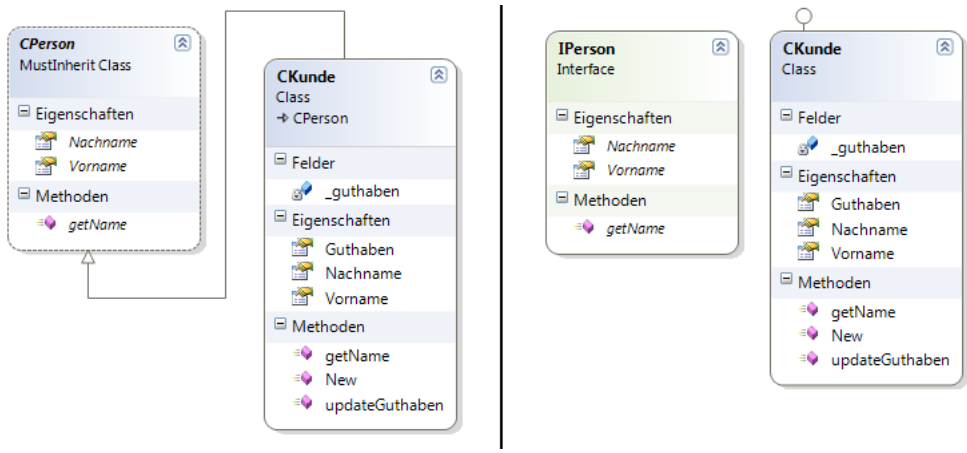
Klassendiagramme

Die erste Variante zeigt das links abgebildete Klassendiagramm. Hier erbt die Klasse *CKunde* von der abstrakten Klasse *CPerson*. Letztere verfügt ausschließlich über abstrakte Klassenmitglieder.

[1] Dies ist allerdings die extremste Form der Implementierungsvererbung, denn es wird de facto keinerlei Code vererbt.

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 in der Laufzeitansicht gezeigt.

Lassen Sie uns mit der ersten Variante beginnen!

Quellcode CPerson

Fügen Sie dem Projekt eine neue Klasse *CPerson* hinzu:

```
Public MustInherit Class CPerson
    Public MustOverride Property Nachname As String
    Public MustOverride Property Vorname As String
    Public MustOverride Function getName() As String
End Class
```

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:

```
Public Class CKunde
    Inherits CPerson

    Private _guthaben As Decimal
```

Die drei von *CPerson* geerbten abstrakten Klassenmitglieder müssen überschrieben werden (wir verwenden selbst implementierende Eigenschaften, die die entsprechenden Zustandsvariablen *_Vorname* und *_Nachname* automatisch im Hintergrund anlegen):

```
    Public Overrides Property Vorname As String
    Public Overrides Property Nachname As String

    Public Overrides Function getName() As String
        Return _Vorname & " " & _Nachname
    End Function
```

Es folgen die normalen Klassenmitglieder:

```
    Public Sub New(vor As String, nach As String)       ' ein Konstruktor
        _Vorname = vor
        _Nachname = nach
    End Sub

    Public ReadOnly Property Guthaben As Decimal
        Get
            Return _guthaben
        End Get
    End Property

    Public Sub updateGuthaben(betrag As Decimal)
        _guthaben += betrag
    End Sub
End Class
```

HINWEIS: Es müssen **alle** geerbten abstrakten Klassenmitglieder überschrieben werden, ansonsten erfolgt eine Fehlermeldung des Compilers.

Quellcode Form1

```
Public Class Form1
```

Zu Beginn wird ein Kunde erzeugt, initialisiert und angezeigt:

```
    Private kunde1 As CKunde
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        kunde1 = New CKunde("Max", "Müller")
        TextBox1.Text = kunde1.Vorname
        TextBox2.Text = kunde1.Nachname
        TextBox3.Text = "10,50"
    End Sub
```

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 Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1.Vorname = TextBox1.Text
```

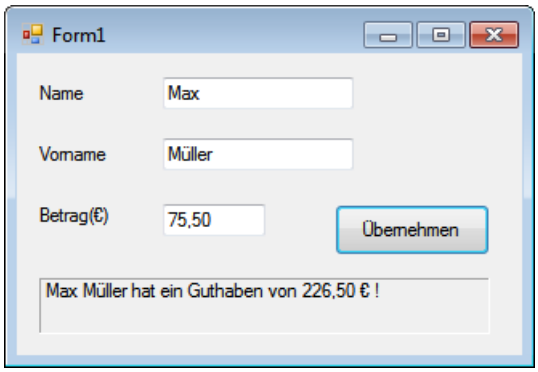
```
        kunde1.Nachname = TextBox2.Text
        Dim betrag As Decimal = Convert.ToDecimal(TextBox3.Text)
        kunde1.updateGuthaben(betrag)
```

Die abgeschlossene Buchung wird mit einem Meldungstext quittiert:

```
        Label1.Text = kunde1.getName() & " hat ein Guthaben von " &
                                          kunde1.Guthaben.ToString("C") & " !"
    End Sub
End Class
```

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 und vereinfachen Sie den Code wie folgt:

```
Public Interface IPerson
    Property Nachname As String
    Property Vorname As String
    Function getName() As String
End Interface
```

Quellcode CKunde

Auch hier sind nur minimale Änderungen erforderlich: An die Stelle von *Inherits* tritt *Implements*. Bei den von *IPerson* geerbten Schnittstellenmitgliedern fehlen die *Overrides*-Modifizierer. Stattdessen werden durch abschließende Kennzeichnung der Methodendeklarationen mit *Implements* und Benennung der Schnittstellenmitglieder Mehrdeutigkeiten vermieden.

```
Public Class CKunde
    Implements IPerson

    Private _guthaben As Decimal
    Public Property Vorname As String Implements IPerson.Vorname
    Public Property Nachname As String Implements IPerson.Nachname

    Public Function getName() As String Implements IPerson.getName
        Return _Vorname & " " & _Nachname
    End Function
...
```

Der restliche Code bleibt unverändert.

Das war es auch schon, denn der Quellcode von *Form1*, in welchem die Klasse CKunde instanziiert und verwendet wird, gleicht bis ins letzte Detail dem seines Vorgängers.

Auch beim Testen des Codes werden Sie keinerlei Veränderungen zum Vorgängerprojekt feststellen.

Vergleichen Sie beide Varianten, so stellen Sie fest, dass die Realisierung mittels Schnittstelle die Transparenz und Übersichtlichkeit des Codes deutlich steigert.

3.12.5 Aggregation und Vererbung gegenüberstellen

Jeder Programmierer hat den Ehrgeiz, mit möglichst wenig Schreibarbeit auszukommen und möglichst viel von seinem Code wieder verwenden zu können. Voraussetzung dafür sind optimale Klassendiagramme, für die es unter dem Aspekt der Wiederverwendbarkeit von Code zwei wesentliche Beziehungen gibt:

- Aggregation/Komposition
- Vererbung

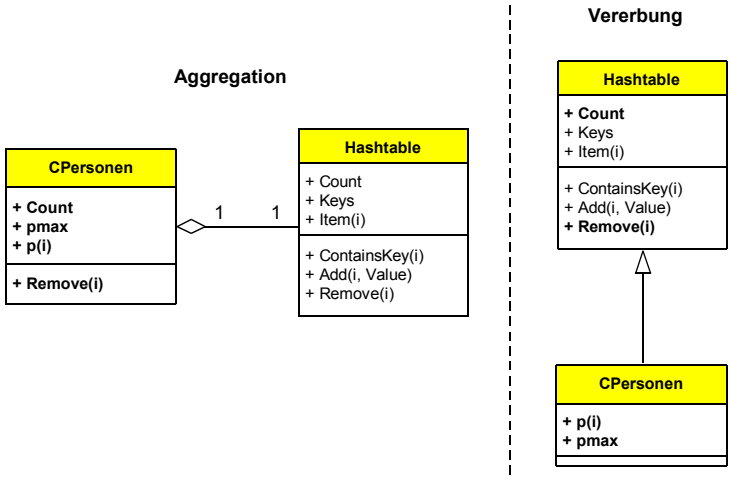
Damit sind wir bereits bei zwei fundamentalen Begriffen der OOP angelangt. Ob wir von *Aggregation* oder *Komposition* sprechen ist in diesem Zusammenhang unerheblich, denn die *Komposition* stellt lediglich die stärkere Form der *Aggregation* dar.

Unter Vererbung ist hier genau genommen die *Implementierungsvererbung* gemeint, denn die unter .NET ebenfalls mögliche *Interfacevererbung* erspart keinerlei Schreibarbeit.

HINWEIS: Sowohl Aggregation/Komposition als auch Vererbung verlangen eine spezifische Herangehensweise bei der Implementierung, die clientseitige Nutzung der Klasse ist aber identisch.

In unserem Beispiel soll dies an Hand einer kleinen Personalverwaltung demonstriert werden, wobei alle Personen in einer Auflistung gespeichert sind, welche die Funktionalität der recht leistungsfähigen *Hashtable*-Klasse nutzt.

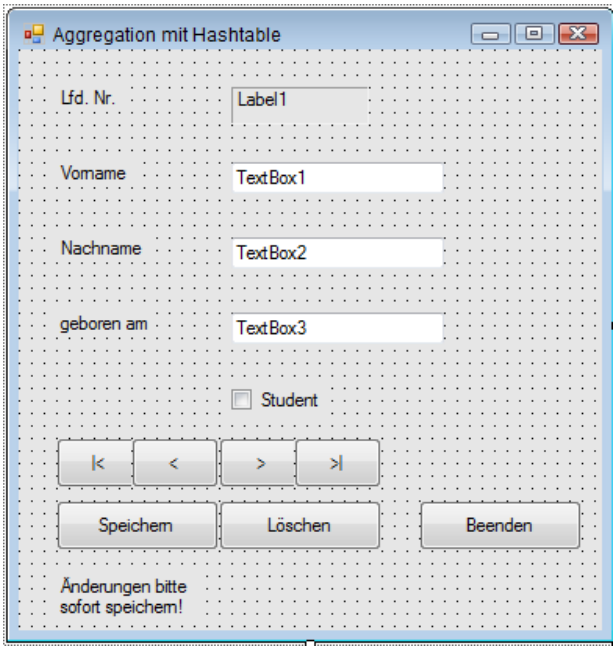
In der folgenden Abbildung sind die Klassendiagramme beider Varianten gegenübergestellt.



Die Klasse *CPersonen* soll über die Standardeigenschaft *p(i)* einen indizierten Lese- und Schreibzugriff auf die Elemente der Auflistung bereitstellen, um damit die *Item*- und die *Add*-Methode von *Hashtable* zu kapseln. Weitere Eigenschaften sind *Count* (Gesamtzahl der abgespeicherten Personen) sowie *pmax* (höchster Index bzw. Schlüsselwert innerhalb der Auflistung). Die von *Hashtable* direkt geerbte *Remove*-Methode ermöglicht das Löschen einer bestimmten Person.

Bedienoberfläche

Das abgebildete Hauptformular bedarf wohl keiner besonderen Erläuterung:



Variante 1: Klassen CPerson und CPersonen mit Aggregation

```
Public Class Form1
```

Der Übersichtlichkeit halber haben wir hier den Code beider Klassen zum *Form1*-Klassenmodul hinzugefügt, Sie könnten ihn aber auch ohne weiteres in ein oder zwei separate Klassenmodule auslagern.

```
    Public Class CPerson
        Public Vorname, Nachname As String
        Public Geburt As Date
        Public student As Boolean
    End Class

    Public Class CPersonen
```

Innerhalb der Klasse *CPersonen* wird die Klasse *Hashtable* instanziiert – das ist Aggregation pur!

```
        Private ht As New Hashtable()
```

Mit einem kleinen Trick, wir definieren eine Eigenschaft *p* als Standardeigenschaft, implementieren wir einen *Quasi-Indexer*[1] für den indizierten Zugriff auf die Personenliste.

HINWEIS: Mittels dieser Standardeigenschaft kann eine *CPersonen*-Collection auf die gleiche Weise wie ein Array indiziert werden!

```
        Default Public Property p(i As Integer) As CPerson
            Get
                If ht.ContainsKey(i) Then
                    Return CType(ht.Item(i), CPerson)
                Else
                    Return Nothing
                End If
            End Get
            Set(ByVal value As CPerson)
                If ht.ContainsKey(i) Then
                    ht.Item(i) = value        ' überschreiben, falls Schlüssel vorhanden
                Else
                    ht.Add(i, value)          ' anhängen, falls Schlüssel noch nicht besetzt
                End If
            End Set
        End Property
```

Das Löschen eines Elements der Auflistung:

```
        Public Sub Remove(i As Integer)
            ht.Remove(i)
        End Sub
    End Class
```

[1] Im Unterschied zu C# erlaubt Visual Basic nicht die direkte Implementierung eines Indexers.

Nachdem die Klassen *CPerson* und *CPersonen* implementiert sind, geht es mit dem eigentlichen Code von *Form1* weiter.

Objekte instanziieren:

```
Private pListe As New CPersonen()     ' die Personenliste
Private person As CPerson             ' die aktuelle Person
```

Zustandsvariablen zum Steuern der Anzeige:

```
Private pos As Integer = 1            ' die aktuelle Position
Private pmax As Integer               ' die max. Anzahl von Personen
```

Die Startaktivitäten:

```
Protected Overrides Sub OnLoad(e As EventArgs)
    Label1.Text = pos.ToString()
    anzeigeLöschen()
    MyBase.OnLoad(e)
End Sub
```

Die Anzeige der aktuellen Person:

```
Private Sub anzeigen()
    Label1.Text = pos.ToString()
    person = pListe(pos)              ' Zugriff wie über Indexer!
    Try
        With person
            TextBox1.Text = .Vorname
            TextBox2.Text = .Nachname
            TextBox3.Text = .Geburt.ToString("dd.MM.yyyy")
            CheckBox1.Checked = .student
        End With
    Catch
        anzeigeLöschen()
    End Try
End Sub
```

Die Hilfsroutine zum Löschen der Anzeige:

```
 Private Sub anzeigeLöschen()
    Label1.Text = String.Empty
    TextBox1.Text = String.Empty : TextBox2.Text = String.Empty
    TextBox3.Text = "00:00:00"
    CheckBox1.Checked = False
End Sub
```

Vorwärts blättern:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    pos += 1
    anzeigen()
End Sub
```

Rückwärts blättern:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        If pos > 1 Then
            pos -= 1
            anzeigen()
        End If
    End Sub
```

Zum Anfang:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        pos = 1
        anzeigen()
    End Sub
```

Zum Ende:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        pos = pmax
        anzeigen()
    End Sub
```

Speichern der aktuellen Person:

```
    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        person = New CPerson()            ' nur Instanzen können hinzugefügt werden!!!
        With person
            .Vorname = TextBox1.Text
            .Nachname = TextBox2.Text
            .Geburt = Convert.ToDateTime(TextBox3.Text)
            .student = CheckBox1.Checked
        End With
        pListe(pos) = person              ' Zugriff wie über Indexer
        If pos > pmax Then pmax = pos     ' max. Schlüsselwert
       anzeigen()
    End Sub
```

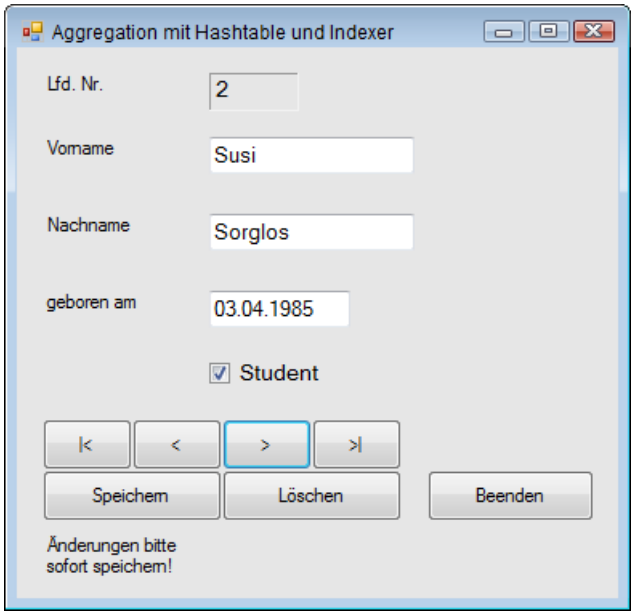
Löschen der aktuellen Person:

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        pListe.Remove(pos)
        anzeigeLöschen()
    End Sub
    ...
End Class
```

Test

Es können beliebig viele Personen eingegeben werden. Der zugeteilte Schlüsselwert entspricht der "Lfd.Nr"-Anzeige. Wenn unter dem Schlüssel bereits eine Person existiert, wird diese überschrieben, anderenfalls neu angelegt.

HINWEIS: Das Abspeichern passiert nicht automatisch beim Weiterblättern, sondern Sie müssen **vor** dem Weiterblättern die *Speichern*-Schaltfläche klicken (ansonsten sind die Änderungen futsch)!



Variante 2: Klasse CPersonen mit Vererbung

Alternativ zur Aggregation können wir die Klasse *CPersonen* auch so implementieren, dass die Eigenschaften/Methoden direkt von der Klasse *Hashtable* "geerbt" werden (siehe Abbildung zu Beginn).

```
Public Class CPersonen
    Inherits Hashtable              ' Vererbung

    Default Public Property p(i As Integer) As CPerson
        Get
            If ContainsKey(i) Then
                Return CType(Item(i), CPerson)
            Else
                Return Nothing
            End If
        End Get
        Set(ByVal value As CPerson)
            If ContainsKey(i) Then
                Item(i) = value             ' überschreiben, falls Schlüssel vorhanden
            Else
                Add(i, value)               ' anhängen, falls Schlüssel noch nicht besetzt
            End If
```

```
            End Set
        End Property
    End Class
```

Vergleichen Sie diesen Code mit der ersten Variante, so stellen Sie fest, dass eine Instanziierung von *Hashtable* nicht mehr erforderlich ist. Stattdessen können die benötigten Eigenschaften und Methoden der Basisklasse direkt aufgerufen werden.

Da dank Implementierungsvererbung alle öffentlichen Eigenschaften/Methoden von *Hashtable* jetzt auch in der Schnittstelle von *CPersonen* verfügbar sind, entfällt auch die Implementierungen der *Remove*-Methode.

Bis auf die vereinfachte Klasse *CPersonen* sind, gegenüber der ersten Variante (Aggregation) keine weiteren Unterschiede festzustellen, der Client "sieht" die gleiche Schnittstelle.

Test

Das Ergebnis ist erwartungsgemäß identisch mit Variante 1.

HINWEIS: Damit der mühsam eingegebene Personalbestand das Ausschalten des Rechners überlebt, ist das Abspeichern in eine Datei erforderlich.

3.12.6 Eine Klasse zur Matrizenrechnung entwickeln

Eine Matrix ist nichts weiter wie der mathematische Begriff für ein Array. In diesem Beispiel soll am Beispiel einer Klasse *CMatrix* die grundlegende Vorgehensweise bei der Entwicklung einer Klasse erläutert werden, die schon etwas anspruchsvoller ist als z.B. eine triviale *CPerson*-Klasse.

Die Schwerpunktthemen sind:

- überladener Konstruktor
- überladene Methoden
- Eigenschaftsmethoden
- Standardeigenschaft als Indexer
- Unterschied zwischen statischen (Shared-) Methoden und Instanzen-Methoden

Die Klasse *CMatrix* soll Funktionalität zur Verfügung stellen, die Sie zur Ausführung von Matrixoperationen benötigen (Addition, Multiplikation...).

Obwohl wir hier nur die Addition implementieren werden, kann die Klasse von Ihnen nach dem gezeigten Muster selbständig um weitere Matrizenoperationen erweitert werden, wie z.B. Multiplikation oder Inversion.

HINWEIS: Wer sich nicht für Mathematik interessiert, kann das Beispiel trotzdem sehr gut verwenden, da der Schwerpunkt auf den verwendeten Programmiertechniken im Zusammenhang mit dem Array-Zugriff liegt!

Quellcode der Klasse CMatrix

Wir beginnen diesmal nicht mit dem Startformular (*Form1*), sondern erweitern zunächst über den Menüpunkt *Projekt|Klasse hinzufügen...* unser Projekt um eine neue Klasse mit dem Namen *CMatrix*.

Die Klasse *CMatrix* verwaltet ein zweidimensionales Array aus *Double*-Zahlen. Die Zustandsvariablen *_rows* und *_cols* speichern die Anzahl der Zeilen und Spalten.

```
Public Class CMatrix
    Private _rows, _cols As Integer
    Private _array(,) As Double
```

Ein neues Array wird über den Konstruktor instanziiert, der in zwei Versionen vorliegt. Falls Sie später *New()* ohne Argument aufrufen, wird eine Matrix mit einem einzigen Element generiert, ansonsten mit den gewünschten Dimensionen.

```
    Sub New()
        MyBase.New()
        _rows = 1
        _cols = 1
        ReDim _array(_rows, _cols)
    End Sub

    Sub New(R As Integer, C As Integer)      ' überladener Konstruktor
        MyBase.New()
        _rows = R
        _cols = C
        ReDim _array(_rows, _cols)
    End Sub
```

Der Zugriff auf die (privaten) Zustandsvariablen *_rows* und *_cols* wird über die Eigenschaften *Rows* und *Cols* gekapselt.

```
    Public Property Rows() As Integer          ' Eigenschaft zum Zugriff auf Zeilenanzahl
        Get
            Return _rows
        End Get
        Set(Value As Integer)
            _rows = Value
        End Set
    End Property

    Public Property Cols() As Integer         ' Eigenschaft zum Zugriff auf Spaltenanzahl
        Get
            Return _cols
        End Get
```

```
        Set(Value As Integer)
            _cols = Value
        End Set
    End Property
```

Der Zugriff auf ein bestimmtes Matrix-Element wird elegant über die Standardeigenschaft realisiert, die hier quasi wie ein Indexer funktioniert:

```
    Default Public Property Cell(row As Integer, col As Integer) As Double
        Get
            Return _array(row, col)
        End Get
        Set(Value As Double)
            _array(row, col) = Value
        End Set
    End Property
```

Die *Add*-Methode akzeptiert entweder ein oder zwei *CMatrix*-Objekte als Parameter, falls Sie nur ein *CMatrix*-Objekt übergeben, wird die aktuelle Instanz der Matrix als zweiter Operand verwendet.

Die erste Überladung der *Add*-Methode ist statisch, sie wird also nicht über einem *CMatrix*-Objekt, sondern direkt über der *CMatrix*-Klasse ausgeführt! Die Methode nimmt beide Operanden (*CMatrix*-Objekte) entgegen und liefert ein *CMatrix*-Objekt zurück.

```
    Public Overloads Shared Function Add(A As CMatrix, B As CMatrix) As CMatrix
        If Not (A.Rows = B.Rows And A.Cols = B.Cols) Then
            Add = New CMatrix()
            Exit Function
        End If
        Dim newMatrix As New CMatrix(A.Rows, A.Cols)
        For row As Integer = 0 To A.Rows - 1
            For col As Integer = 0 To A.Cols - 1
                newMatrix(row, col) = A(row, col) + B(row, col)
            Next
        Next
        Return newMatrix
    End Function
```

Obige Methode wird mit einer leeren "Verlegenheitsmatrix" verlassen, wenn beide Operanden nicht die gleichen Dimensionen aufweisen sollten.

Bei der zweiten Überladung handelt es sich um eine normale Instanzen-Methode, sie nimmt als Parameter nur ein einziges *CMatrix*-Objekt entgegen. Der zweite Operand ist naturgemäß die aktuelle *CMatrix*-Instanz, die diese Methode aufruft.

```
    Public Overloads Function Add(A As CMatrix) As CMatrix
        If Not (A.Rows = MyClass.Rows And A.Cols = MyClass.Cols) Then
            Return New CMatrix()
            Exit Function
        End If
        Dim newMatrix As New CMatrix(MyClass.Rows, MyClass.Cols)
```

```
        For row As Integer = 0 To MyClass.Rows - 1
            For col As Integer = 0 To MyClass.Cols - 1
                newMatrix(row, col) = A(row, col) + MyClass.Cell(row, col)
            Next
        Next
        Return newMatrix
    End Function

End Class
```

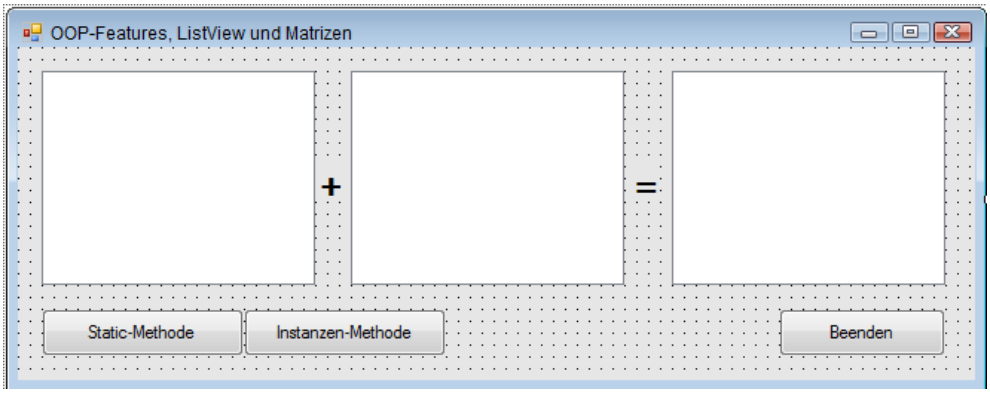
Der Unterschied zwischen statischen- und Instanzen-Methode dürfte Ihnen so richtig erst beim Sichten des Codes von *Form1* klar werden, wo beide Überladungen aufgerufen werden.

Hier ein Vorgriff auf den Code von *Form1*:

```
    Dim A, B, C As CMatrix
    ...
    C = CMatrix.Add(A, B)               ' Aufruf der  statischen Methode
    C = A.Add(B)                        ' Aufruf der Instanzen-Methode
```

Oberfläche

Wir benötigen drei *ListView*-Komponenten und drei Schaltflächen. Setzen Sie folgende zwei Eigenschaften für jede *ListView*: *View* = *Details* und *GridLines* = *True*.



Quellcode von Form1

```
Public Class Form1
```

Wir verwenden für beide Schaltflächen einen gemeinsamen Eventhandler:

```
    Private Sub Button_Click(sender As Object, e As EventArgs) _
                                                Handles Button1.Click, Button2.Click
```

Unser Beispiel benutzt Matrizen mit 9 Zeilen und 6 Spalten:

```
        Const rows As Integer = 9    ' Anzahl Zeilen
        Const cols As Integer = 6    ' Anzahl Spalten
```

Zufallszahlengenerator instanziieren:

```
Dim rnd As New System.Random()
```

Die Matrix *A* instanziieren, mit Zufallszahlen füllen und anzeigen (man beachte den bequemen Zugriff über den Indexer!):

```
 Dim A As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To A.Rows - 1
    For j As Integer = 0 To A.Cols - 1
        A(i, j) = rnd.Next(100)           ' Zugriff auf Matrixelement über Indexer!
    Next
Next
showListView(A, ListView1)                ' Anzeige in linker ListView
```

Gleiches geschieht mit Matrix *B*:

```
Dim B As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To B.Rows - 1
    For j As Integer = 0 To B.Cols - 1
        B(i, j) = rnd.Next(100)
    Next
Next
showListView(B, ListView2)                ' Anzeige in mittlerer ListView
```

Die resultierende Matrix *C* berechnen wir – in Abhängigkeit vom geklickten *Button* – mit der ersten oder mit der zweiten Überladung der *Add*-Methode.

HINWEIS: Beide Überladungen der *Add*-Methode leisten absolut das Gleiche, nur die Aufruf-Syntax ist unterschiedlich!

```
Dim C As CMatrix
If CType(sender, Button) Is Button1 Then
    C = CMatrix.Add(A, B)                 ' Aufruf Shared-Methode
Else
    C = A.Add(B)                          ' Aufruf Instanzen-Methode
End If
showListView(C, ListView3)                ' Anzeige in rechter ListView
End Sub
```

Der Anzeigeroutine *showListView* werden ein *CMatrix*-Objekt und eine *ListView*-Komponente übergeben:

```
Private Sub showListView(M As CMatrix, lv As ListView)
    With lv
        .Clear()
```

Alle Spalten erzeugen und beschriften:

```
        .Columns.Add("", 20, HorizontalAlignment.Right)     ' linke (leere) Randspalte
        For j As Integer = 0 To M.Cols - 1
```

Spaltennummerierung und Formatierung in Kopfzeile:

```
                .Columns.Add(j.ToString, 30, HorizontalAlignment.Right)
            Next
```

Alle Zeilen erzeugen, beschriften und Zellen füllen:

```
            For i As Integer = 0 To M.Rows - 1
```

Pro Zeile ein *ListViewItem*, Zeilennummerierung in linke Randspalte eintragen:

```
                Dim item As New ListViewItem(i.ToString)
                For j As Integer = 0 To M.Cols - 1
```

Alle Zellen füllen (pro Zelle ein *SubItem*):

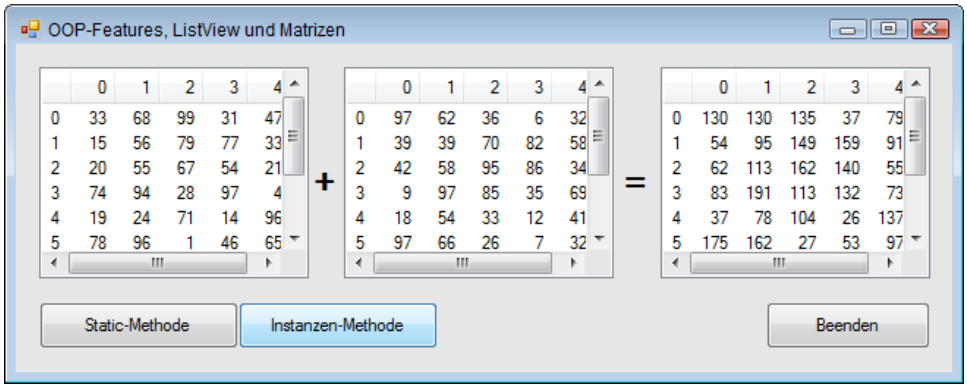
```
                    item.SubItems.Add(M(i, j).ToString)
                Next
```

Zeile zur *ListView* hinzufügen:

```
               .Items.Add(item)
            Next
        End With
    End Sub
End Class
```

Test

Nach Programmstart werden die beiden ersten Matrizen mit Zufallszahlen zwischen 0 und 100 gefüllt. Ob Sie dann *Button1* oder *Button2* klicken ist völlig egal, in beiden Fällen wird die resultierende Summenmatrix mit dem richtigen Ergebnis gefüllt:



Bemerkung

Das Resultat einer Matrix-Operation ist immer eine neue Matrix, wenn allerdings beide Matrizen inkompatibel sind, wird eine leere Matrix zurückgegeben. Alternativ könnte in einem solchen Fall auch eine Ausnahme innerhalb der Methoden erzeugt werden (*Throw New System.ArgumentException()*).

3.12.7 Rechner für komplexe Zahlen

Auch mit dieser Anwendung wollen wir nicht nur die Lösung eines mathematischen Problems zeigen, sondern (was viel wichtiger ist) grundlegendes Handwerkszeug des .NET-Programmierers demonstrieren:

- Sinnvolle Auslagerung von Quellcode in Klassen, um das Verständnis der OOP zu vertiefen,
- Prinzip der Operatorenüberladung in Visual Basic,
- Strukturierung des Codes der Benutzerschnittstelle nach dem EVA-Prinzip (Eingabe – Verarbeitung – Ausgabe).

Doch ehe wir mit der Praxis beginnen, scheint ein kurzer Abstieg in die Untiefen der Mathematik unumgänglich.

Was sind komplexe Zahlen?

Eine besondere Bedeutung haben komplexe Zahlen beispielsweise in der Schwingungslehre und in der Wechselstromtechnik, einem bedeutenden Teilgebiet der Elektrotechnik.

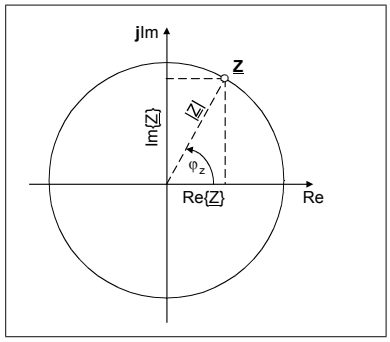
Zur Darstellung einer komplexen Zahl **Z** bieten sich zwei Möglichkeiten an:

- Kartesische Koordinaten (Real-/Imaginärteil)
- Polarkoordinaten (Betrags-/Winkeldarstellung)

Die folgende Tabelle zeigt eine Zusammenstellung der Umrechnungsformeln:

Kartesische Koordinaten	Polarkoordinaten
$\underline{Z} = \mathrm{Re}\{\underline{Z}\} + j\,\mathrm{Im}\{\underline{Z}\}$	$\underline{Z} = \lvert\underline{Z}\rvert\, e^{j\varphi_z}$
Realteil: $\mathrm{Re}\{\underline{Z}\} = \lvert\underline{Z}\rvert \cos\varphi_z$	Betrag: $\lvert\underline{Z}\rvert = \sqrt{(\mathrm{Re}\{Z\})^2 + (\mathrm{Im}\{Z\})^2}$
Imaginärteil: $\mathrm{Im}\{\underline{Z}\} = \lvert\underline{Z}\rvert \sin\varphi_z$	Phasenwinkel: $\varphi_Z = \arctan\frac{\mathrm{Im}\{Z\}}{\mathrm{Re}\{Z\}}$

Am besten lassen sich diese Zusammenhänge am Einheitskreis erläutern, wobei **Z** als Punkt in der komplexen Ebene erscheint:



Die kartesische Form eignet sich besonders gut für die Ausführung von Addition und Subtraktion:

Mit

$$\mathbf{Z_1 = a_1 + jb_1} \quad \text{und} \quad \mathbf{Z_2 = a_2 + jb_2}$$

ergibt sich

$$\mathbf{Z_1 + Z_2 = a_1 + a_2 + j(b_1 + b_2)} \quad \text{bzw.} \quad \mathbf{Z_1 - Z_2 = a_1 - a_2 + j(b_1 - b_2)}$$

Andererseits bevorzugt man für Multiplikation und Division die Zeigerform:

Mit

$$\mathbf{Z_1 = c_1 \cdot e^{j\varphi_1}} \quad \text{und} \quad \mathbf{Z_2 = c_2 \cdot e^{j\varphi_2}}$$

erhalten wir

$$\mathbf{Z_1 \cdot Z_2 = c_1 \cdot c_2 \cdot e^{j(\varphi_1 + \varphi_2)}} \quad \text{bzw.} \quad \mathbf{Z_1/Z_2 = c_1/c_2 \cdot e^{j(\varphi_1 - \varphi_2)}}$$

Für die Angabe des Phasenwinkels hat man die Wahl zwischen Radiant (Bogenmaß) und Grad. Für die gegenseitige Umrechnung gilt die Beziehung

$$\varphi(Rad) = \frac{\pi}{180}\varphi(Grad)$$

HINWEIS: Die Maßeinheit "Grad" wird aufgrund ihrer Anschaulichkeit vom Praktiker für die Ein- und Ausgabe bevorzugt, während "Radiant" für interne Berechnungen günstiger ist.

Programmierung der Klasse CComplexN

Öffnen Sie ein neues Projekt vom Typ Windows Forms-Anwendung. Das Startformular *Form1* lassen Sie zunächst unbeachtet liegen, denn der routinierte .NET-Programmierer kapselt seinen Code in Klassen anstatt ihn einfach zum Formularcode hinzuzufügen.

Die zweckmäßige Aufteilung einer praktischen Problemstellung in verschiedene Klassen und die Definition der Abhängigkeiten ist sicherlich der schwierigste Part der OOP und erfordert einige Übung und Routine, bis das dazu erforderliche abstrakte Denken schließlich zur Selbstverständlichkeit wird[1].

Die hier vorgeschlagene Lösung benutzt die Klasse *CComplexN*, welche eine komplexe Zahl repräsentiert. Diese Klasse speichert in den Zustandsvariablen *Re* und *Im* (die in unserem Fall gleichzeitig Eigenschaften sind) den Wert der komplexen Zahl in Kartesischen Koordinaten. Die beiden anderen Eigenschaften (*Len* und *Ang*) repräsentieren dieselbe Zahl in Polar-Koordinaten. Allerdings werden *Len* und *Ang* nicht direkt in den Objekten gespeichert, sondern in so genannten *Eigenschaftenmethoden* (*property procedures*) jeweils aus *Re* und *Im* berechnet.

Über das Menü *Projekt|Klasse hinzufügen...* erstellen Sie den Rahmencode der Klasse.

[1] Die UML (Unified Modelling Language) stellt dazu spezielle Werkzeuge bereit.

```
Public Class CComplexN
```

Die beiden öffentlichen Zustandsvariablen *Re* und *Im* bilden das "Gedächtnis" der Klasse und können quasi wie Eigenschaften benutzt werden[1]:

```
    Public Re, Im As Double                  ' Real- und Imaginärteil
```

Ein Konstruktor (den Rahmencode können Sie sich von der IDE erzeugen lassen) setzt die Zustandsvariablen auf ihre Anfangswerte:

```
    Public Sub New(r As Double, i As Double)
        Re = r : Im = i
    End Sub
```

Die "intelligente" Eigenschaftsmethode *Ang* berechnet den Phasenwinkel aus den Zustandsvariablen *Re* und *Im*:

```
    Public Property Ang() As Double
        Get
                Dim g As Double = 0
                If Re <> 0 Then
                g = 180 / Math.PI * Math.Atan(Im / Re)
                If Re < 0 Then g += 180
                Else
                If Im <> 0 Then
                      If Im > 0 Then g = 90
                Else
                     g = -90
                 End If
               End If
               Return g
            End Get
            Set(value As Double)
                Dim b, l As Double
                b = value * Math.PI / 180
                l = Math.Sqrt(Re * Re + Im * Im)
                Re = l * Math.Cos(b)        ' neuer Realteil
                Im = l * Math.Sin(b)        ' neuer Imaginärteil
          End Set
        End Property
```

Die Eigenschaft *Len* ermittelt den Betrag (die Länge des Zeigers) aus *Re* und *Im*:

```
        Public Property Len() As Double
          Get
             Return Math.Sqrt(Re * Re + Im * Im)
          End Get
          Set(value As Double)
             Dim b As Double = Math.Atan(Im / Re)
```

[1] Die Verwendung öffentlicher Zustandsvariabler als Eigenschaften ist zwar nicht der sauberste, in unserem Fall aber der effektivste Weg.

```
            Re = value * Math.Cos(b)
            Im = value * Math.Sin(b)
         End Set
      End Property
```

Besonders interessant sind die folgenden drei (statischen) Methoden, welche die Operatorenüberladungen für Addition, Multiplikation und Division neu definieren.

Der "+"-Operator erhält eine neue Bedeutung, er addiert jetzt zwei komplexe Zahlen:

```
      Public Shared Operator +(a As CComplexN, b As CComplexN) As CComplexN
         Dim z As New CComplexN(0, 0)
         z.Re = a.Re + b.Re
         z.Im = a.Im + b.Im
         Return z
      End Operator
```

Der "*"-Operator multipliziert zwei komplexe Zahlen:

```
      Public Shared Operator *(a As CComplexN, b As CComplexN) As CComplexN
         Dim z As New CComplexN(0, 0)
         z.Re = a.Re * b.Re - a.Im * b.Im
         z.Im = a.Re * b.Im + a.Im * b.Re
         Return z
      End Operator
```

Der "/"-Operator dividiert zwei komplexe Zahlen:

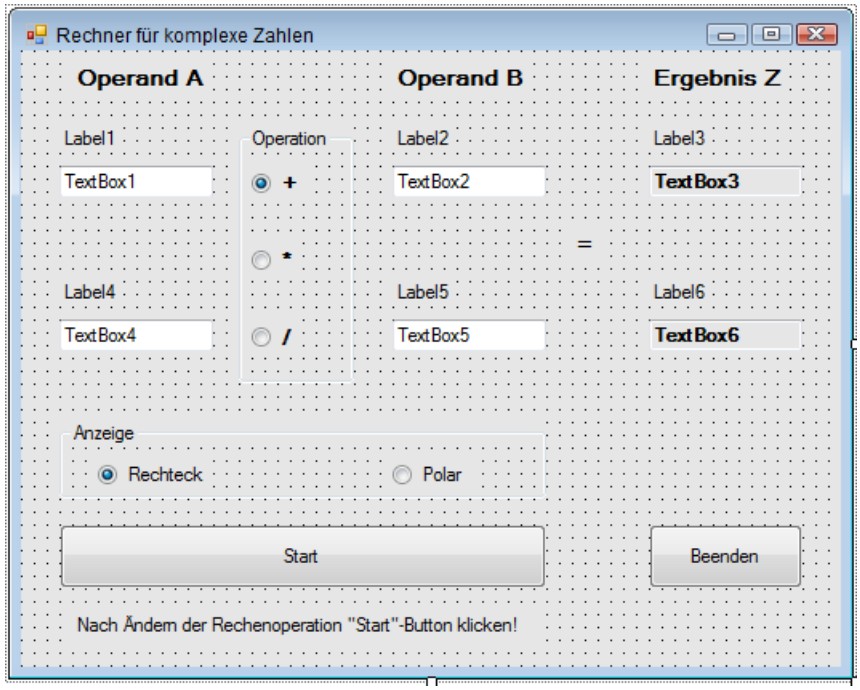
```
       Public Shared Operator /(a As CComplexN, b As CComplexN) As CComplexN
          Dim z As New CComplexN(0, 0)
          z.Re = (a.Re * b.Re + a.Im * b.Im) / (b.Re * b.Re + b.Im * b.Im)
          z.Im = (a.Im * b.Re - a.Re * b.Im) / (b.Re * b.Re + b.Im * b.Im)
          Return z
      End Operator
End Class
```

HINWEIS: Vielleicht sticht Ihnen bereits jetzt ein gravierender Unterschied zur klassischen "Geradeausprogrammierung" ins Auge: Spezielle Methoden zur Umrechnung zwischen Kartesischen- und Polarkoordinaten sind Fehlanzeige, da ein Objekt vom Typ *CComplexN* beide Darstellungen bereits als Eigenschaften kapselt!

Bedienoberfläche für Testprogramm

Um uns von der Funktionsfähigkeit der entwickelten Klassen zu überzeugen, brauchen wir ein kleines Testprogramm, das die Ein- und Ausgabe von komplexen Zahlen und die Auswahl der Rechenoperation sowie der Koordinatendarstellung ermöglicht.

Wir verwenden dazu das bereits vorhandene Startformular *Form1*, das wir entsprechend der folgenden Abbildung gestalten.



HINWEIS: Es kann nicht schaden, wenn Sie *ReadOnly* für *TextBox3* und *TextBox6* auf *True* und *TabStop* auf *False* setzen, da Sie diese beiden rechten Felder nur zur Ergebnisanzeige brauchen.

Quellcode für Testprogramm

Das an legendäre DOS-Zeiten erinnernde EVA-Prinzip (Eingabe, Verarbeitung, Anzeige) hat auch unter .NET nichts von seiner grundlegenden Bedeutung eingebüßt.

Der clientseitige Quellcode entspricht vom prinzipiellen Ablauf her der klassischen Geradeausprogrammierung, ist allerdings deutlich übersichtlicher und problemnäher, denn wir arbeiten mit drei Objektvariablen, die bereits komplexe Zahlen sind und nicht mit einer Vielzahl skalarer Variablen!

```
Public Class Form1
```

Die benötigten Objektvariablen:

```
    Private A As New CComplexN(1, 1)          ' Operand A
    Private B As New CComplexN(1, 1)          ' Operand B
    Private Z As New CComplexN(0, 0)          ' Ergebnis Z
```

Unter Berücksichtigung der eingestellten Anzeigeart (Rechteck- oder Polarkoordinaten) liest die folgende Methode die Werte aus der Eingabemaske in die Objekte:

```
Private Sub Eingabe()
    If RadioButton4.Checked Then   ' Rechteck-Koordinaten
        A.Re = Convert.ToDouble(TextBox1.Text)
        B.Re = Convert.ToDouble(TextBox2.Text)
        A.Im = Convert.ToDouble(TextBox4.Text)
        B.Im = Convert.ToDouble(TextBox5.Text)
    Else                          ' Polar-Koordinaten
        A.Len = Convert.ToDouble(TextBox1.Text)
        B.Len = Convert.ToDouble(TextBox2.Text)
        A.Ang = Convert.ToDouble(TextBox4.Text)
        B.Ang = Convert.ToDouble(TextBox5.Text)
    End If
End Sub
```

Die Verarbeitungsroutine führt die gewünschte Rechenoperation mit den bekannten Symbolen für Addition, Multiplikation und Division aus. Dazu werden die in der Klasse *CComplexN* definierten Operatorenüberladungen benutzt:

```
Private Sub Verarbeitung()
         If RadioButton1.Checked Then Z = A + B      ' Addition
         If RadioButton2.Checked Then Z = A * B      ' Multiplikation
         If RadioButton3.Checked Then Z = A / B      ' Division
End Sub
```

Als Pendant zur *Eingabe*-Methode sorgt die Methode *Ausgabe* für die Anzeige von *A*, *B* und *Z*, wozu auch die Anpassung der Beschriftung der Eingabefelder gehört:

```
Private Sub Anzeige()
    If RadioButton4.Checked Then          ' Anzeige in Rechteck-Koordinaten
        Label1.Text = "Realteil A"
        Label2.Text = "Realteil B"
        Label3.Text = "Realteil Z"
        Label4.Text = "Imaginärteil A"
        Label5.Text = "Imaginärteil B"
        Label6.Text = "Imaginärteil Z"
        TextBox1.Text = A.Re.ToString()     ' Anzeige Realteil A
        TextBox4.Text = A.Im.ToString()     ' Anzeige Imaginärteil A
        TextBox2.Text = B.Re.ToString()     ' Anzeige Realteil B
        TextBox5.Text = B.Im.ToString()     ' Anzeige Imaginärteil B
        TextBox3.Text = Z.Re.ToString()     ' Anzeige Realteil Z
        TextBox6.Text = Z.Im.ToString()     ' Anzeige Imaginärteil Z
    Else                                  ' Anzeige in Polarkoordinaten
        Label1.Text = "Betrag A"
        Label2.Text = "Betrag B"
        Label3.Text = "Betrag Z"
        Label4.Text = "Winkel A"
        Label5.Text = "Winkel B"
        Label6.Text = "Winkel Z"
```

```
            TextBox1.Text = A.Len.ToString()    ' Anzeige Betrag A
            TextBox4.Text = A.Ang.ToString()    ' Anzeige Winkel A
            TextBox2.Text = B.Len.ToString()    ' Anzeige Betrag B
            TextBox5.Text = B.Ang.ToString()    ' Anzeige Winkel B
            TextBox3.Text = Z.Len.ToString()    ' Anzeige Betrag Z
            TextBox6.Text = Z.Ang.ToString()    ' Anzeige Winkel Z
        End If
    End Sub
```

Die *Start*-Schaltfläche:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Eingabe()
        Verarbeitung()
        Anzeige()
    End Sub
```

Nach Umschaltung zwischen Rechteck- und Polarkoordinaten muss die Anzeige aktualisiert werden:

```
    Private Sub RadioButton4_CheckedChanged(sender As Object,
                        e As EventArgs) Handles RadioButton4.CheckedChanged
        Anzeige()
    End Sub
...
```

Programmtest

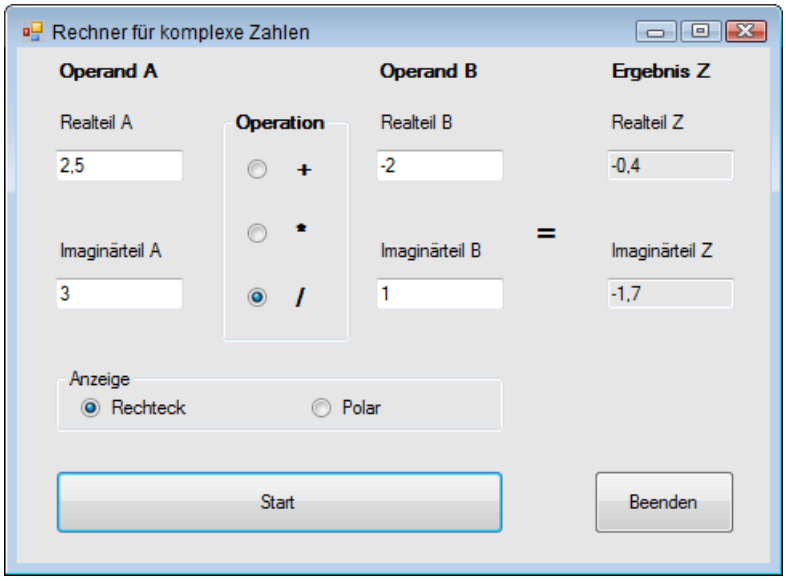
Wenn zum Beispiel die Aufgabe

(2.5 + 3j) / (-2 + j)

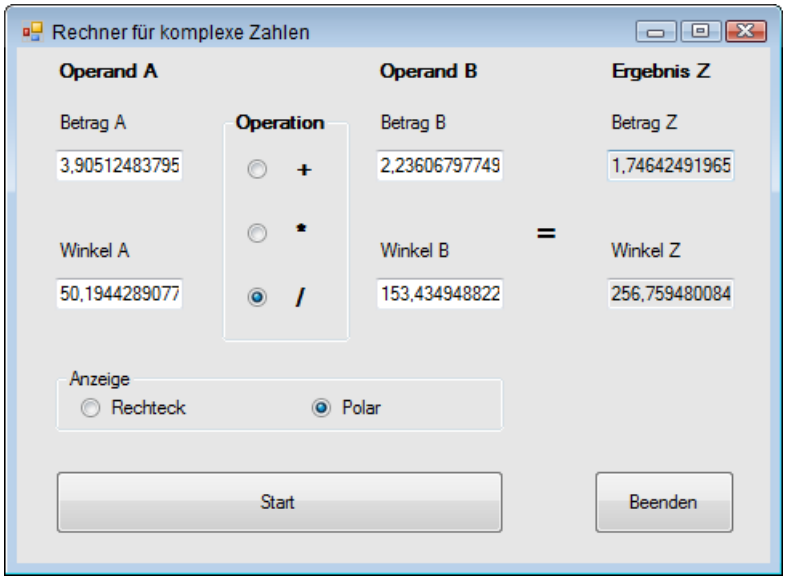
gelöst werden soll, so stellen Sie zunächst die Anzeige auf "Rechteck" ein. Geben Sie dann links oben den Realteil (2,5) und darunter den Imaginärteil (3) des ersten Operanden ein. Analog dazu geben Sie rechts oben den Realteil (-2) und darunter den Imaginärteil (1) des zweiten Operanden ein. Abschließend klicken Sie auf die gewünschte Operation (/).

Nach Betätigen der *Start*-Taste erscheint als Ergebnis die komplexe Zahl -0.4 -1.7j (siehe folgende Abbildung).

Die äquivalenten Polarkoordinaten liefern für das gleiche Beispiel einen Zeiger mit der Länge von ca. 1.746 und einem Winkel von ca. 256.76 Grad.



HINWEIS: Wenn Sie die Anzeige zwischen Rechteck- und Polarkoordinaten umgeschaltet haben, müssen Sie anschließend die *Start*-Schaltfläche klicken!



Bemerkungen

- Die Vorteile eines gut strukturierten, objektorientierten Programms liegen bekanntermaßen in der leichteren Lesbarkeit des Quellcodes ("sprechender" Code) und in der besseren Wartbarkeit und Erweiterungsfähigkeit.
- Beim Arbeiten mit Visual Studio informiert Sie die Intellisense stets aktuell über die vorhandenen Klassenmitglieder und deren Signaturen.
- Bereits mit .NET 4.0 wurde die Klasse *System.Numerics.Complex* eingeführt. Alle Versuche der Autoren, diese Klasse als Ersatz für *CComplexN* zu verwenden und den "Rechner für komplexe Zahlen" damit zu realisieren, scheiterten am unbefriedigenden und praxisfremden Programmiermodell der *Complex*-Klasse. So sind die Eigenschaften *Real* und *Imaginary* schreibgeschützt und nur über den Konstruktor zuweisbar, die Polarkoordinaten hingegen können nur über eine statische Methode gesetzt werden. Der Code wird dadurch unnötig aufgebläht und verliert an Transparenz. Wir haben deshalb auf die Anwendung dieser Klasse verzichtet und bevorzugen weiterhin unsere "Eigenproduktion" *CComplexN*.

HINWEIS: Wer mit der systemeigenen Klasse *Complex* dennoch experimentieren möchte, muss in der Regel vorher einen Verweis auf die *System.Numerics.dll* hinzufügen.

3.12.8 Formel-Rechner mit dem CodeDOM

Jeder, der in einer Mathematik-Ausbildung steht oder im Bereich wissenschaftlich-technischer Anwendungen arbeitet, hat sicher schon vor der Aufgabe gestanden, Berechnungen von Formelausdrücken durchzuführen, sei es um eine Werteliste zu erstellen oder um eine Diagramm auszudrucken.

Wer jetzt befürchtet, dafür erst einen aufwändigen Formelparser entwickeln zu müssen, den können wir beruhigen, denn unter .NET erlaubt das *Code Document Object Model* aus dem Namensraum *System.CodeDOM* eine verblüffend einfache Realisierungsmöglichkeit: Sie können den Quellcode einer .NET-Programmiersprache zur Laufzeit "zusammenbasteln", kompilieren und ausführen! Aus der so erzeugten Assembly kann mittels Reflexion die "zusammengebastelte" Funktion aufgerufen und das Ergebnis ausgewertet werden!

HINWEIS: Ein Rechner nach diesem Prinzip stellt bezüglich seiner Leistungsfähigkeit die bekannten Windows-Taschenrechner weit in den Schatten!

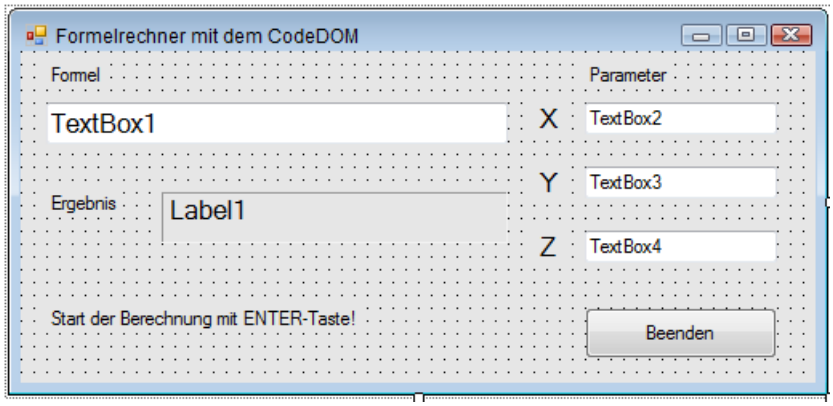
In welcher Sprache Sie die zu berechnende Formel zusammenbauen ist egal, Voraussetzung ist lediglich das Vorhandensein eines zum Compiler passenden *CodeDomProvider*s. Im vorliegenden Fall haben wir uns, wen wundert's, für VB entschieden. Wir hätten aber auch den C#-Provider nehmen können, für den Endanwender ist aber die VB-Syntax einfacher zu verstehen, scheitern doch viele bereits an der peniblen Groß-/Kleinschreibung von C#.

HINWEIS: Ganz abgesehen von seinem Nutzen als universeller Formelrechner bietet dieses Beispiel eine eindrucksvolle Demonstration des Prinzips und der Leistungsfähigkeit des Reflection-Mechanismus von .NET.

Das Grundprinzip des Formelrechners soll zunächst an einer auf das Wesentliche beschränkten Variante demonstriert werden.

Entwurf Bedienoberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und erstellen Sie ein Formular mit folgendem Aussehen:



Bei der Gestaltung der Benutzerschnittstelle (*Form1*) haben Sie viel Spielraum, sodass obige Abbildung lediglich als Vorschlag zu verstehen ist. Wir können sogar auf eine Ergebnis-Schaltfläche ("=") verzichten, wie sie bei "normalen" Taschenrechnern üblich ist. Stattdessen werden wir die Berechnung durch einfaches Betätigen der Enter-Taste starten.

Außerdem gönnen wir uns noch drei weitere *TextBox*en, um auch Parameter in die Formel einbauen zu können (es empfiehlt sich, dazu die *KeyPreview*-Eigenschaft des Formulars auf *True* zu setzen).

Die Klasse CCalculator

Fügen Sie zum Projekt eine neue Klasse mit dem Namen *CCalculator* hinzu. Die Klasse stellt einzig und allein die statische Methode *Calc()* bereit, welcher der zu berechnende Ausdruck als String zu übergeben ist. Der Rückgabewert (*Double*) entspricht dem Ergebnis der Berechnung.

Die *Calc()*-Methode erzeugt den Quellcode für ein gültiges VB-Modul mit einer Klasse, die eine ganz einfache Funktion (ebenfalls mit dem Namen *Calc*) zur Berechnung dieses Ausdrucks kapselt. Der Code wird kompiliert und ausgeführt. Um den Code dem VB-Compiler zu übergeben, kommt das CodeDOM (*Code Document Object Model*) zum Einsatz, mit dem sich aus einer Anwendung heraus Programmcode erzeugen lässt. Nach dem Kompilieren wird mittels Reflection auf die erzeugte Assembly zugegriffen und der Ausdruck berechnet.

```
Imports System.CodeDom.Compiler
Imports System.Reflection

Public Class CCalculator
```

Zwischenspeichern der Assembly und der Verweise:

```
    Private Shared ass As Assembly
    Private Shared aClass As Type
    Private Shared aMethode As MethodInfo
    Private Shared obj As Object
```

Der Berechnungsfunktion wird der Formelausdruck als String übergeben:

```
    Public Shared Function Calc(expr As String) As Double
        If expr.Length = 0 Then Return 0.0
```

Im Formelausdruck werden die Dezimalkommas durch Dezimalpunkte ersetzt:

```
        expr = expr.Replace(",", ".")
```

Compilerparameter definieren:

```
        Dim opt As New CompilerParameters(Nothing, String.Empty, False)
        opt.GenerateExecutable = False
        opt.GenerateInMemory = True
```

Den zu kompilierenden VB-Quellcode müssen wir natürlich noch zeilenweise zusammenbauen, mittendrin findet sich unser zu berechnender Ausdruck. Durch die Anweisung *Imports System.Math* können wir mathematische Funktionen wie *Sin* ... auch ohne vorangestellten Namespace schreiben:

```
        Dim src As String = "Imports System.Math" & vbCrLf &
                            "Public Class Calculate" & vbCrLf &
                                "Public Function Calc() As Double" & vbCrLf &
                                    "Return " & expr & vbCrLf &
                                "End Function" & vbCrLf &
                            "End Class" & vbCrLf
```

Nun kann unser VB-Quellcode kompiliert werden:

```
        Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(opt, src)
```

Auf eine Fehlerauswertung sollte nicht verzichtet werden:

```
        If res.Errors.Count > 0 Then
            Dim errors As String = String.Empty
            For Each cerr As CompilerError In res.Errors
                errors = errors & cerr.ToString() & vbCrLf
            Next
            ass = Nothing
            expr = String.Empty
            Throw New ApplicationException(errors)
        End If
```

Die vom Compiler erzeugte Assembly kann nun ermittelt und mit dem *Reflection*-Mechanismus ausgewertet werden:

```
        ass = res.CompiledAssembly
```

Die interne Klasse aus der Assembly "herausziehen":

```
        aClass = ass.GetType("Calculate")
```

Jetzt kommen wir auch an die interne *Calc*-Methode heran:

```
        aMethod = aClass.GetMethod("Calc")
```

Eine Instanz der internen Klasse erzeugen, die interne *Calc*-Methode aufrufen und das Ergebnis zurück liefern:

```
        obj = Activator.CreateInstance(aClass)
        Return Convert.ToDouble(aMethod.Invoke(obj, Nothing))
    End Function

End Class
```

Quellcode Form1

```
Public Class Form1
```

Die zentrale Anlaufstelle nach Änderung der Eingabewerte ist der Aufruf dieser Methode:

```
    Private Sub Berechnung()
```

Den Formelausdruck zuweisen:

```
        Dim str As String = TextBox1.Text.ToUpper()
```

Die Parameter X, Y, Z direkt in den Formelausdruck einbauen:

```
        str = str.Replace("X", TextBox2.Text).Replace("Y", TextBox3.Text). _
                                                          Replace("Z", TextBox4.Text)
```

Start der Berechnung (eine Instanziierung der Klasse *CCalculator* kann entfallen, da lediglich ein statischer Methodenaufruf erfolgt). Aufgrund der vielen möglichen Compilerfehler bei Syntaxverstößen wird der entscheidende Methodenaufruf in einer Fehlerbehandlung gekapselt:

```
        Try
            Dim res As Double = CCalculator.Calc(str)
            str = res.ToString()
```

Um das Dezimaltrennzeichen einheitlich als Punkt darzustellen, wandeln wir im Ergebnisstring das Komma einfach in einen Punkt um:

```
            Label1.Text = str.Replace(",", ".")
        Catch ex As Exception
            Label1.Text = String.Empty
            MessageBox.Show(ex.Message)
        End Try
    End Sub
```

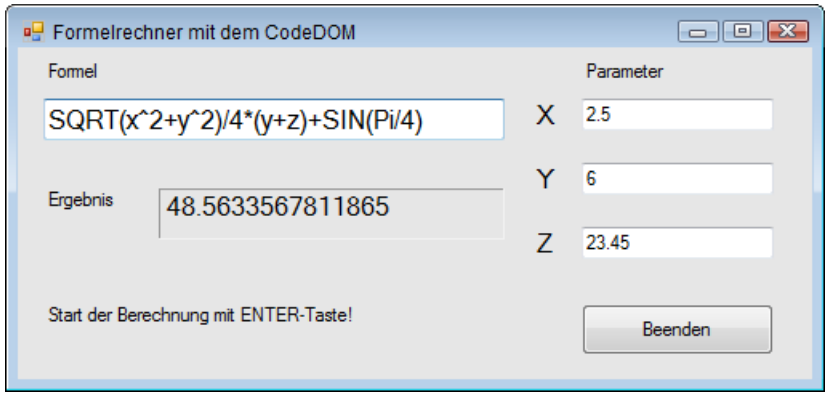
Alle vier *TextBox*en verwenden den folgenden gemeinsamen *KeyPress*-Eventhandler, er sorgt dafür, dass die Berechnung mittels Enter-Taste gestartet wird:

```
    Private Sub TextBox_KeyPress(sender As Object, e As KeyPressEventArgs) _
            Handles TextBox1.KeyPress, TextBox4.KeyPress, TextBox3.KeyPress, TextBox2.KeyPress
        If e.KeyChar = ChrW(Keys.Enter) Then
            Berechnung()
            e.Handled = True
        End If
    End Sub
    ...
End Class
```

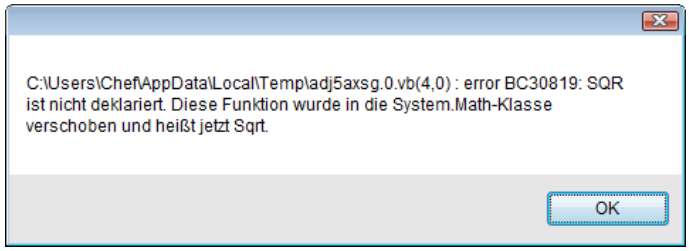
Test

Geben Sie einen beliebig komplizierten bzw. verschachtelten arithmetischen Ausdruck ein (mit oder ohne Parameter *x*, *y*, *z*). Grundlage ist die VB-Syntax, d h. auch, die Groß-/Kleinschreibung ist ohne Bedeutung.

Starten Sie die Berechnung mit der *Enter*-Taste!



Bei syntaktischen Verstößen erfolgen in der Regel recht ausführliche Fehlermeldungen. Das Beispiel in der folgenden Abbildung zeigt die Meldung, wenn versehentlich der VB-Quadratwurzel-Operator *SQRT* mit *SQR* verwechselt wurde:



Bemerkungen

- Die Klasse *CCalculator* ist ausbaufähig, denn sie kann nicht nur einen einzigen Ausdruck, sondern auch einen kompletten Algorithmus berechnen, in welchem weitere Funktionen aufgerufen werden können. In diesem Fall empfiehlt sich ein mehrzeiliges Textfeld, in das der VB-Code einzugeben ist. Vorher ist auf das Vorhandensein der *Return*-Anweisung zu prüfen, sodass diese nur im Bedarfsfall (wie in unserem Beispiel) per Programmcode hinzugefügt werden muss.
- Wollen Sie für den zu berechnenden Ausdruck nicht die VB-, sondern die C#-Syntax verwenden, muss natürlich ein anderer Quellcode "zusammengebaut" werden und Sie müssen den entsprechende C#-Codeprovider instanziieren. Bei der Eingabe der Berechnungsformel wäre dann penibel auf die Groß-/Kleinschreibung zu achten.

3.12.9 Einen Funktionsverlauf grafisch darstellen

In der Basisversion des Formelrechners wurde gezeigt, wie man mittels CodeDOM beliebig komplizierte mathematische Formeln auswerten kann. Will man allerdings eine grafische Auswertung (Funktionsdiagramm) vornehmen, so sind in der Regel einige hundert aufeinanderfolgende Aufrufe der Formel bei schrittweise sich ändernden Parametern erforderlich. Damit stößt das Verfahren an seine Grenzen, denn das wiederholte Kompilieren und Auswerten würde auch bei einem schnellen Rechner unzumutbar viel Zeit verbrauchen.

Da es nicht sinnvoll ist, bei geänderten Parametern aber gleichem Formelausdruck die Assembly immer wieder erneut zu kompilieren, speichern wir die entsprechenden Verweise auf die einmal erzeugte Assembly intern (im Arbeitsspeicher) ab und greifen bei Bedarf darauf zu.

Wir wollen die Vorgehensweise anhand einer bescheidenen Grafik für die bekannte Spaltfunktion *sin(x)/x* erläutern.

Oberfläche

Das nackte Startformular (*Form1*), ausgestattet mit einer einzigen *TextBox*, genügt! Weisen Sie der *TextBox* die *Text*-Eigenschaft "*Sin(x) / x*" zu (Leerzeichen und Groß-/Kleinschreibung spielen dabei keine Rolle).

Die Klasse CCalculatorX

Die Änderungen gegenüber der Klasse *CCalculator* sind durch Fettdruck hervorgehoben:

```
Imports System.CodeDom.Compiler
Imports System.Reflection

Public Class CCalculatorX
    Private Shared ass As Assembly
    Private Shared aClass As Type
    Private Shared aMethod As MethodInfo
    Private Shared obj As Object
```

```
    Private Shared expression As String

    Public Shared Function Calc(expr As String, x As Double) As Double
        If expr.Length = 0 Then Return 0.0
        expr = expr.Replace(",", ".")
```

Ein erneutes Kompilieren ist nur dann erforderlich, wenn sich der Formelausdruck *expr* geändert hat:

```
        If expr <> expression Then
            expression = expr
            Dim opt As New CompilerParameters(Nothing, String.Empty, False)
            opt.GenerateExecutable = False
            opt.GenerateInMemory = True

            Dim src As String = "Imports System.Math" & vbCrLf &
            "Public Class Calculate" & vbCrLf &
                "Public Function Calc(X As Double) As Double" & vbCrLf &
                    "Return " & expr & vbCrLf &
                "End Function" & vbCrLf &
            "End Class" & vbCrLf

            Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(
                                                                    opt, src)
            If res.Errors.Count > 0 Then
                Dim errors As String = String.Empty
                For Each cerr As CompilerError In res.Errors
                    errors = errors & cerr.ToString() & vbCrLf
                Next
                ass = Nothing
                expr = String.Empty
                Throw New ApplicationException(errors)
            End If
            ass = res.CompiledAssembly
            aClass = ass.GetType("Calculate")
            aMethod = aClass.GetMethod("Calc")
            obj = Activator.CreateInstance(aClass)
        End If
```

Falls sich *expr* nicht geändert hat, genügt die Übergabe des neuen Wertes für den Parameter *x*:

```
        Return Convert.ToDouble(aMethod.Invoke(obj, New Object() {x}))
    End Function

End Class
```

Quellcode Form1

```
Public Class Form1
```

Wir überschreiben die *OnPaint*-Methode des Formulars, dadurch wird auch nach vorübergehendem Verdecken des Fensters das Diagramm automatisch neu erstellt:

```
    Protected Overrides Sub OnPaint(e As System.Windows.Forms.PaintEventArgs)
```

Um den Quellcode überschaubar zu halten, haben wir die Einstellungen des Koordinatensystems für das Diagramm der Funktion *y = sin(x) / x* optimiert:

```
        Dim x As Single = -40
        Dim y As Single = 0
        Dim xold As Single = x
        Dim yold As Single = y
        Dim scalex As Single = Me.Width / (2 * x)
        Dim scaley As Single = -(Me.Height - 20) / 2
        Dim incr As Single = 80.0F / Me.Width
        Dim p As New Pen(Color.Black, 2)
```

Graphics-Objekts, auf welches das Diagramm gezeichnet werden kann:

```
        Dim g As Graphics = e.Graphics
```

Koordinatensystem verschieben:

```
        g.TranslateTransform(Me.Width / 2, Me.Height / 2)
```

Koordinatenachsen zeichnen:

```
        g.DrawLine(Pens.Red, -Me.Width, 0, Me.Width, 0)
        g.DrawLine(Pens.Red, 0, -Me.Height, 0, Me.Height)
```

Anzeige optimieren:

```
        g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias
```

Die einzelnen Funktionswerte berechnen und zeichnen:

```
        For i As Integer = 0 To Me.Width - 1
            If x = 0 Then x += 0.000001F
            y = Convert.ToSingle(CCalculatorX.Calc(TextBox1.Text, x))
            x += incr
            g.DrawLine(p, x * scalex, y * scaley, xold * scalex, yold * scaley)
            xold = x
            yold = y
        Next
```

Die folgende Anweisung nicht vergessen, sonst wird *Paint*-Event nicht ausgelöst:

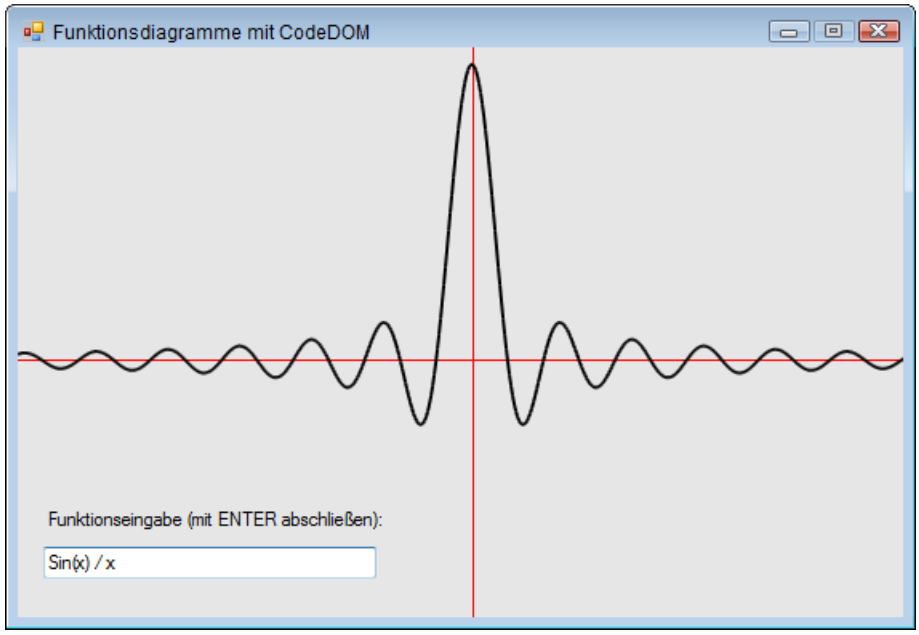
```
        MyBase.OnPaint(e)
    End Sub
```

Der folgende Eventhandler sorgt für die Umwandlung eines Dezimalkommas in einen Dezimalpunkt und für das Neuzeichnen des Formulars nach Betätigen der Enter-Taste:

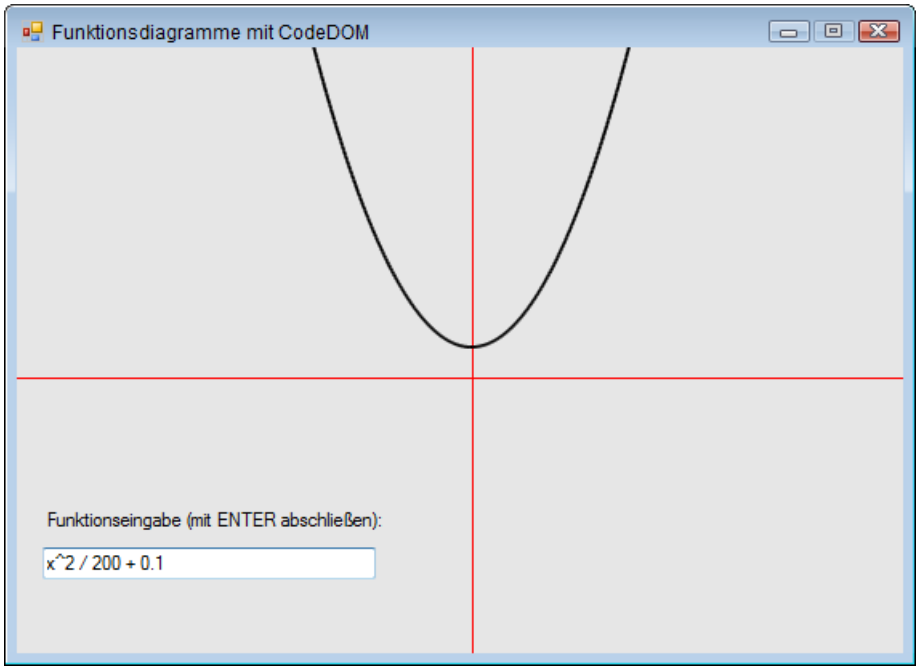
```
    Private Sub TextBox1_KeyPress(sender As Object, e As KeyPressEventArgs) _
                                                        Handles TextBox1.KeyPress
        If e.KeyChar = "," Then
            e.KeyChar = "."
        Else
            If e.KeyChar = ChrW(Keys.Enter) Then
                Me.Refresh()
                e.Handled = True
            End If
        End If
    End Sub
End Class
```

Test

Gleich nach Programmstart erscheint das Diagramm der Funktion *sin(x)/x*. Die Grafik wird auch nach vorübergehendem Verdecken des Fensters in Windeseile wieder aufgebaut, da nur einmal kompiliert werden muss.



Ändern Sie den Formelausdruck und schließen Sie mit der *Enter*-Taste ab. Unter der Voraussetzung, dass das starre Koordinatensystem eine geeignete Darstellung zulässt, lassen sich interessante Experimente durchführen:



Bemerkungen

- Das Programm lässt sich durch Hinzufügen weiterer Bedienelemente für Inkremente, Maßstabsfaktoren, Skalenteilung etc. so verfeinern, dass eine optimale Darstellung nahezu beliebiger Funktionen ermöglicht wird.
- Die spezialisierte Klasse *CCalculatorX* hat gegenüber der Klasse *CCalculator* den Vorteil, dass sie wiederholte Berechnungen mit einem geänderten Parameter *x* gestattet, ohne dass dazu die Assembly erneut erzeugt werden müsste. Das bedeutet einen erheblichen Performancegewinn.
- Der Klasse *CCalculator* bleibt hingegen der Vorteil der universellen Verwendbarkeit für beliebige Ausdrücke mit beliebig vielen Parametern, sie eignet sich deshalb besonders für einmalig auszuführende komplizierte Berechnungen.

3.12.10 Sortieren mit IComparable/IComparer

Haben Sie eine *ArrayList* oder eine generische *List(Of T)* von Typen, wie Strings oder Integers, die bereits *IComparer* unterstützen, so können Sie dieses Array oder die Liste ohne irgendeine explizite Referenz auf *IComparer* sortieren. In diesem Fall werden die Elemente des Arrays automatisch in die standardmäßige Implementierung von *IComparer* gecastet, eine *Sort*-Methode ist also bereits "eingebaut".

Haben Sie aber nutzerdefinierte Objekte, so müssen Sie selbst entweder eines oder beide der Interfaces *IComparable* oder *IComparer* implementieren.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie eine *ListBox* (*Name=lb*) und zwei Schaltflächen im Formular:

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <ListBox Name="lb"  Grid.Column="0" ItemsSource="{Binding}"/>
    <StackPanel Grid.Column="1">
        <Button Content="IComparable" Click="Button_Click_1" />
        <Button Content="IComparer"  Click="Button_Click_2" />
    </StackPanel>
</Grid>
```

Klasse CStudent

Fügen Sie zum Projekt eine neue Klasse *CStudent* hinzu, die das *IComparable*-Interface implementiert:

```
Public Class CStudent
 Implements IComparable(Of CStudent)

 Public Property Nummer As Integer
 Public Property Vorname As String
 Public Property Nachname As String

 Public Sub New(numm As Integer, vor As String, nach As String)
     Nummer = numm
     Vorname = vor
     Nachname = nach
 End Sub
```

Die alphabetische Sortierung nach dem Nachnamen wird als Standardvergleich festgelegt:

```
    Public Function CompareTo(stud As CStudent) As Integer _
                                Implements IComparable(Of CStudent).CompareTo
        Return Me.Nachname.CompareTo(stud.Nachname)
    End Function
```

Um die Datenbindung zu vereinfachen überschreiben wir die *ToString*-Methode:

```
    Public Overrides Function ToString() As String
        Return Nummer.ToString() & " ,  " & Nachname & " , " & Vorname
    End Function
End Class
```

MainWindow.xaml.vb

Im Hauptfenster implementieren wir zunächst eine Methode, die eine Studentenliste erzeugt und initialisiert:

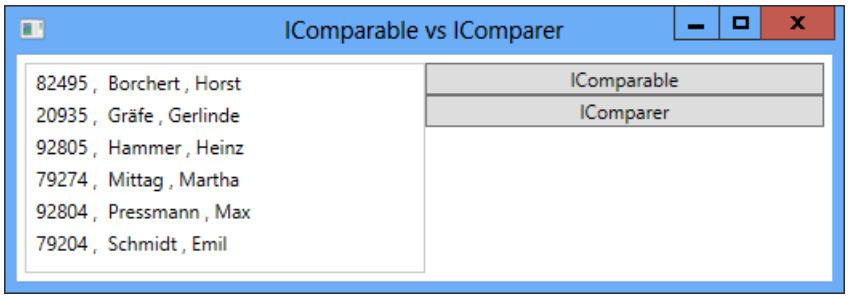
```
Class MainWindow

    Private Function getStudenten() As List(Of CStudent)
        Dim studenten As New List(Of CStudent)()
        studenten.Add(New CStudent(82495, "Horst", "Borchert"))
        studenten.Add(New CStudent(20935, "Gerlinde", "Gräfe"))
        studenten.Add(New CStudent(79274, "Martha", "Mittag"))
        studenten.Add(New CStudent(79204, "Emil", "Schmidt"))
        studenten.Add(New CStudent(92804, "Max", "Pressmann"))
        studenten.Add(New CStudent(92805, "Heinz", "Hammer"))
        Return studenten
    End Function
```

Im *Click*-Event des ersten *Button*s wird die Liste (entsprechend der Implementierung von *CompareTo* in *CStudent*) sortiert und angezeigt:

```
     Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Dim studenten = getStudenten()
        studenten.Sort()
        lb.DataContext = studenten
     End Sub
...
```

Test von IComparable

Bei Klick auf die Schaltfläche *IComparable* erscheint die Liste alphabetisch nach den Nachnamen sortiert:



Bemerkungen zu IComparable

- Wenn eine Klasse das *IComparable* Interface implementiert, müssen wir auch die Methode *CompareTo(T)* implementieren. In dieser Methode können wir unseren Sortieralgorithmus definieren (Standardvergleich). In unserem Beispiel haben wir die Liste in alphabetischer Reihenfolge sortiert.

- Wir verwenden *IComparable(Of T)*, wenn die Klasse über einen Standardvergleich verfügen soll. Das Sortierkriterium muss also bereits bekannt sein, bevor wir mit der Implementierung der Klasse beginnen. In unserem Beispiel mussten wir deshalb vorher entscheiden, dass wir nach dem Nachnamen und nicht nach der Studentennummer sortieren wollen. Es gibt aber Situationen, wo wir nicht nur den Standardvergleich, sondern mehrere Sortierkriterien benötigen.
- Um letztgenanntes Problem zu lösen, stellt .NET ein spezielles Interface *IComparer(Of T)* bereit, welches über eine Methode *Compare()* verfügt, die zwei Objektparameter X, Y entgegennimmt und ein Integer zurückgibt.

Klasse CSortNummer

Fügen Sie zum Projekt eine Klasse *CSortNummer* hinzu, welche das *IComparer*-Interface implementiert:

```
Public Class CSortNummer
    Implements IComparer(Of CStudent)
```

Eine Zustandsvariable bestimmt die Sortierung in aufsteigender (*True*) bzw. absteigender (*False*) Folge:

```
    Private _auf As Boolean
```

Wir übergeben im Konstruktor einen Parameter, der die gewünschte Sortierfolge spezifiziert.

```
    Public Sub New(auf As Boolean)
      _auf = auf
    End Sub
```

Die Implementierung des Sortierkriteriums erfolgt in der *Compare*-Methode:

```
  Public Function Compare(a As CStudent, b As CStudent) As Integer _
                                          Implements IComparer(Of CStudent).Compare
    If _auf Then         ' aufsteigend
        If (a.Nummer > b.Nummer) Then
            Return 1
        ElseIf a.Nummer < b.Nummer Then
            Return -1
        Else
            Return 0
        End If
    Else                 ' absteigend
        If a.Nummer < b.Nummer Then
            Return 1
        ElseIf a.Nummer > b.Nummer Then
            Return -1
        Else
            Return 0
        End If
```

```
        End If
    End Function
End Class
```

Ergänzung MainWindow.xaml.vb

Wir ergänzen den Codebehind, um die zweite Sortiervariante zu testen:

```
    ...
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Dim studenten = getStudenten()
```

Wir wollen die Sortierfolge umkehren:

```
        Dim son As New CSortNummer(False)
```

Die Liste wird nun nach fallenden Nummern sortiert:

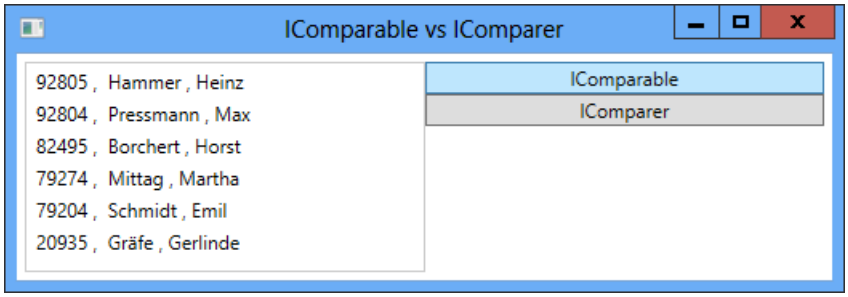
```
        studenten.Sort(son)
```

Noch eine Bemerkung zu obiger Zeile: Die IntelliSense wird Ihnen zwei Überladungen der *Sort*-Methode anbieten, eine davon ist die bereits implementierte Standardsortierung (*IComparable*), die andere die in *CSortNummer* definierte zusätzliche Sortierung, die wir hier ausgewählt haben.

```
        lb.DataContext = studenten
    End Sub
    ...
```

Test von IComparer

Die Liste ist jetzt nach fallenden Nummern sortiert:



Bemerkungen zu IComparer

- Wir verwenden *IComparer* dann, wenn wir ein anderes Sortierkriterium als den von der Klasse bereitgestellten Standardvergleich benötigen (oder mehrere Vergleichskriterien).
- Pro zusätzliches Sortierkriterium ist eine Klasse erforderlich, die das *IComparer*-Interface (*Compare*-Methode) implementiert.
- *IComparer* funktioniert auch dann, wenn keine Standardsortierung existiert, also wenn *IComparable* nicht implementiert ist.

3.12.11 Objektbäume in generischen Listen abspeichern

Die generische *List*-Klasse hat gegenüber der altbackenen *ArrayList* gravierende Vorteile, so entfallen vor allem die umständlichen Typkonvertierungen. Nicht nur einfache Objekte, sondern auch komplette Objektbäume können mittels *BinaryFormatter* bequem serialisiert bzw. deserialisiert werden, um die Daten dauerhaft zu sichern.

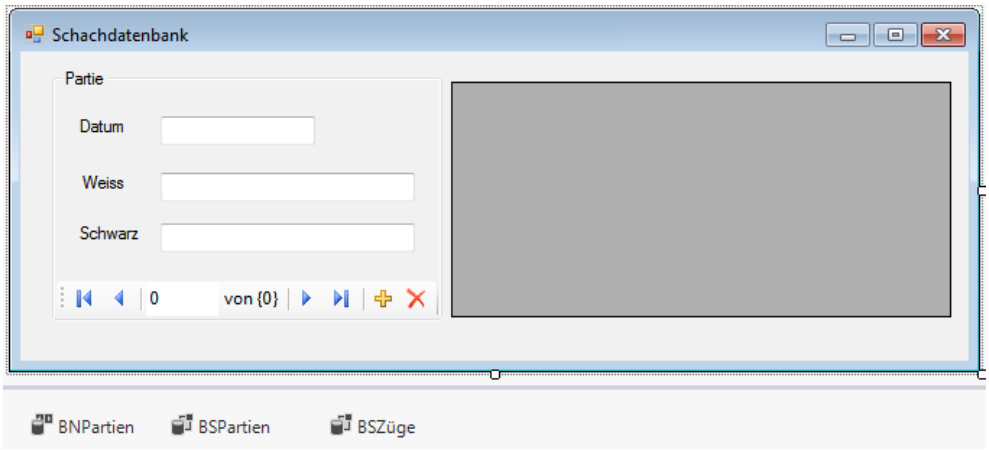
Wir demonstrieren dies am Beispiel einer ausbaufähigen Schachdatenbank, in der Sie beliebig viele Schachpartien abspeichern können[1].

HINWEIS: Das vorliegende Beispiel verwendet keine Datenbank, sondern eine einfache Binärdatei als Speichermedium!

Oberfläche

Starten Sie Visual Studio und öffnen Sie eine neue Windows Forms-Anwendung. Auf dem Startformular *Form1* findet links eine *GroupBox* ihren Platz. In diese setzen Sie drei *TextBox*-Steuerelemente und einen *BindingNavigator*, der am unteren Rand angedockt wird (*Dock* = *Bottom*) und den Sie in *BNPartien* umbenennen.

Rechts platzieren Sie eine *DataGridView*-Komponente. Fügen Sie noch zwei *BindingSource*-Komponenten hinzu, die Sie in *BSPartien* und *BSZüge* umbenennen. Verbinden Sie nun die *BindingSource*-Eigenschaft von *BNPartien* mit *BSPartien*.



Über das Menü *Projekt/Klasse hinzufügen...* erweitern Sie das Projekt um die Klassen *CZug*, *CPartie* und *CSchachDB*, die beide am Objektbaum beteiligt sind. Alle werden mit dem *Serializable*-Attribut ausgestattet, da wir den Objektbaum nicht nur im Arbeitsspeicher ablegen, sondern auch auf der Festplatte sichern wollen.

[1] Um den Sinn dieses Beispiels zu verstehen, muss man kein Schachspieler sein.

Die Klasse CZug

Diese Klasse stellt die Blattebene unseres Objektbaums dar und repräsentiert einen Doppelzug nebst Kommentar (einfachheitshalber sind alle Eigenschaften selbst implementierend):

```
<Serializable> Public Class CZug
    Public Property Weiss As String       ' z.B. "e4" oder Le7, Se6, 0-0, Lb4+, c:d5, S:a4, ...
    Public Property Schwarz As String
    Public Property Kommentar As String   ' z.B. "schwerer Fehler!"
End Class
```

Die Klasse CPartie

```
<Serializable> Public Class CPartie
```

Eine generische Liste speichert alle Züge einer Partie:

```
    Private _züge As List(Of CZug)
```

Im Konstruktor wird die (zunächst leere) Zugliste erstellt:

```
    Sub New()
        _züge = New List(Of CZug)
    End Sub
```

Der Zugriff auf die Zugliste:

```
    Public Property Züge As IList(Of CZug)
        Get
            Return _züge
        End Get
        Set(value As IList(Of CZug))
            _züge = value
        End Set
    End Property
```

Einige selbst implementierende Eigenschaften:

```
   Public Property Datum As DateTime             ' Datum der Partie
   Public Property Weiss As String               ' Name des Weiss-Spielers
   Public Property Schwarz As String             ' Name des Schwarz-Spielers
End Class
```

Die Klasse CSchachDB

Dies ist die Wurzelklasse unseres Objektbaums.

```
<Serializable> Public Class CSchachDB
```

Die folgende (generische) Liste speichert alle Partien der Datenbank:

```
   Private _partien As IList(Of CPartie)
```

Im Konstruktor wird die (zunächst leere) Partienliste erzeugt:

```
Sub New()
     _partien = New List(Of CPartie)
End Sub
```

Der Zugriff auf die Liste der Partien:

```
Public Property Partien() As IList(Of CPartie)    ' Liste der Partien
    Get
        Return _partien
    End Get
    Set(value As IList(Of CPartie))
        _partien = value
    End Set
End Property

End Class
```

Die Klasse CPersistenz

Damit nach dem Ausschalten des Rechners nicht alle Daten auf Nimmerwiedersehen verschwunden sind, stellt diese Klasse die Methoden *saveObject* und *loadObject* bereit, die sich um die Datenpersistenz kümmern.

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Public Class CPersistenz
```

Aufgabe dieser Methode ist es, den kompletten Objektbaum zu serialisieren und als Datei abzuspeichern:

```
    Public Shared Sub saveObject(o As Object, pfad As String)
        Dim fs As New FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None)
        Dim bf As New BinaryFormatter()
        bf.Serialize(fs, o)
        fs.Close()
    End Sub
```

Die folgende Methode lädt die Datei zurück in den Arbeitsspeicher und rekonstruiert den Objektbaum:

```
   Public Shared Function loadObject(pfad As String) As Object
        Dim fs As New FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read)
        Dim bf As New BinaryFormatter()
        Dim o As Object = bf.Deserialize(fs)
        fs.Close()
        Return o
    End Function
End Class
```

Die Klasse Form1

```
Public Class Form1
```

Der komplette Objektbaum:

```
    Private _schachDB As New CSchachDB()
```

Der Standort der Datenbankdatei:

```
    Private Const PFAD As String = "SchachDB.dat"
```

Der Programmstart:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Zunächst wird versucht, die Datenbankdatei zu laden, im Fehlerfall entsteht eine neue leere Datei:

```
        Try
            _schachDB = CType(CPersistenz.loadObject(PFAD), CSchachDB)
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
```

Verbinden der *Partien*-Liste mit der entsprechenden *BindingSource*-Komponente:

```
        BSPartien.DataSource = _schachDB.Partien
```

Anbinden der Steuerelemente:

```
        TextBox1.DataBindings.Add("Text", BSPartien, "Datum", True)
        TextBox2.DataBindings.Add("Text", BSPartien, "Weiss", True)
        TextBox3.DataBindings.Add("Text", BSPartien, "Schwarz", True)
```

Anbinden des Datengitters an die Liste der Züge:

```
        DataGridView1.DataSource = BSZüge
    End Sub
```

Erzeugen Sie einen Eventhandler für das *CurrentChanged*-Ereignis der *BindingSource* der Partienliste:

```
    Private Sub BSPartien_CurrentChanged(sender As Object, e As EventArgs) _
                                        Handles BSPartien.CurrentChanged
```

Beim Navigieren zu einer anderen Partie wird die *BindingSource* der Zügeliste umgeklemmt:

```
        Dim partie As CPartie = CType(BSPartien.Current, CPartie)
        BSZüge.DataSource = partie.Züge
    End Sub
```

Beim Schließen des Formulars versuchen wir, den Objektbaum in der Datei abzuspeichern:

```
    Private Sub Form1_FormClosing(sender As Object, e As FormClosingEventArgs) _
                                                Handles Me.FormClosing
        Try
            CPersistenz.saveObject(_schachDB, PFAD)
        Catch ex As Exception
```

```
            MessageBox.Show(ex.Message)
        End Try
    End Sub
```

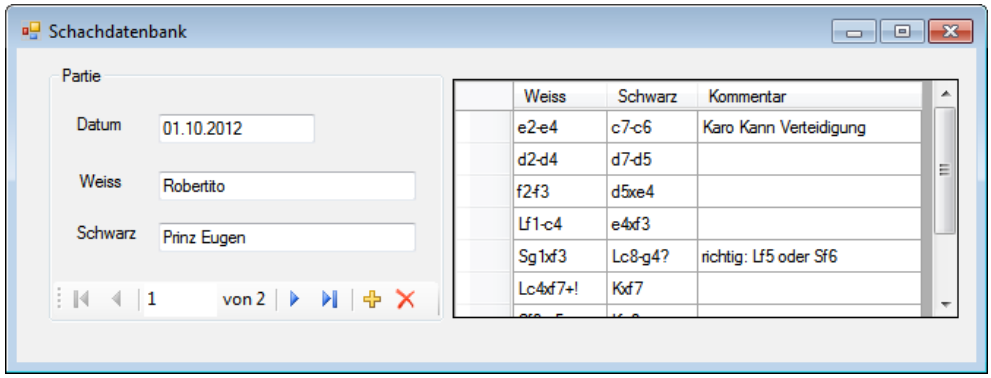
Damit Sie nicht leichtfertig eine komplette Schachpartie mitsamt allen mühsam eingegebenen Zügen löschen, soll nach Klick auf die *Delete*-Schaltfläche des *BindingNavigator*s zunächst eine Sicherheitswarnung erscheinen:

```
    Private Sub BindingNavigatorDeleteItem2_Click(sender As Object, e As EventArgs) _
                                                Handles BindingNavigatorDeleteItem2.Click
        If MsgBox("Soll die Partie wirklich gelöscht werden?", MsgBoxStyle.YesNo,
                                              "Sicherheitsabfrage") = MsgBoxResult.Yes Then
            BSPartien.RemoveCurrent()
        End If
    End Sub

End Class
```

HINWEIS: Obige Methode funktioniert nur dann, wenn Sie die *DeleteItem*-Eigenschaft des *BindingNavigator*s auf *None* (keine) gesetzt haben.

Test

Lassen Sie sich nicht davon irritieren, dass nach Programmstart zunächst ein Meldungsfenster erscheint, welches Sie auf die noch nicht vorhandene Datei hinweist.



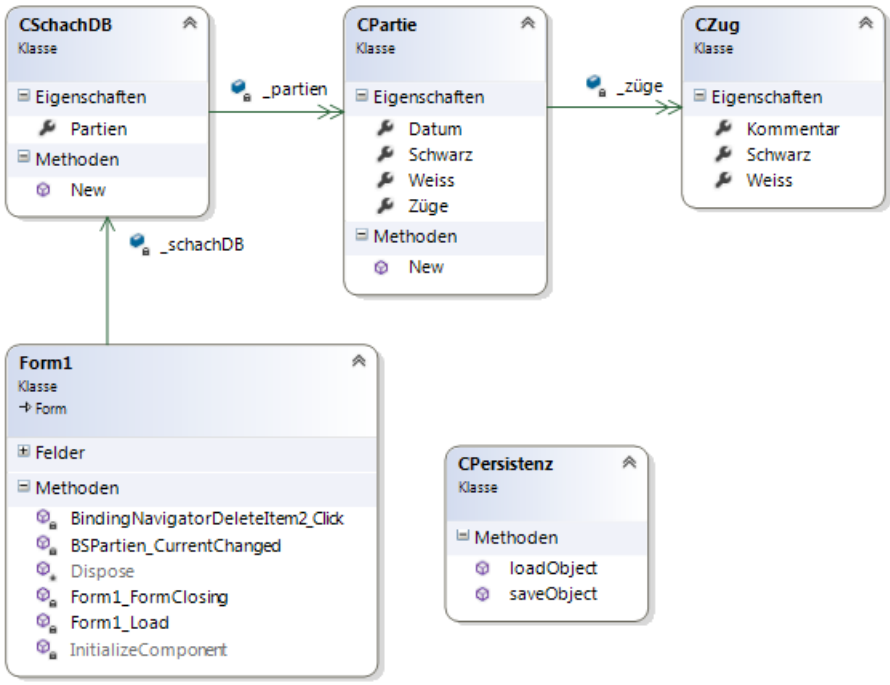
Klicken Sie zunächst auf die "+"-Schaltfläche im *BindingNavigator*, um eine neue Schachpartie mit dem Datum[1] und den Namen der Spieler anzulegen. Danach können Sie auf bekannte Weise im Datengitter nacheinander die einzelnen Züge eintippen.

[1] Haben Sie ein ungültiges Datum eingegeben, so lässt sich das Formular erst dann wieder schließen, wenn Sie diesen Fehler korrigiert haben (generische Listen sind typsicher!).

Nach Schließen des Formulars werden alle Partien automatisch in der im Anwendungsverzeichnis abgelegten Datei *Schachpartien.dat* gespeichert und stehen nach erneutem Programmstart wieder zur Verfügung.

Bemerkungen

- Für den Schachinteressenten ergeben sich mannigfaltige Erweiterungsmöglichkeiten, wie zum Beispiel die Suche nach Partien mit einem bestimmten Gegner oder nach gleichen Eröffnungszügen. Ziemlich aufwändig, aber auch sehr praktisch ist eine grafische Brettdarstellung, mittels welcher man die Züge per Drag&Drop eingeben kann.
- Einen guten Überblick über die Programmstruktur und die Beziehungen zwischen den Klassen liefert das **Klassendiagramm**, welches man sich im Projektmappen-Explorer generieren lassen kann (Kontextmenü *Klassendiagramm anzeigen*). Die Auflistungszuordnungen (Assoziationen) der Felder *_partien* und *_züge* sind an den doppelten Pfeilspitzen erkennbar. Man erhält sie, wenn man zunächst auf das entsprechende Feld in der Klasse klickt und dann im Kontextmenü *Als Auflistungszuordnung anzeigen* wählt. Die folgende Abbildung zeigt nur eine von vielen möglichen Versionen des Klassendiagramms.



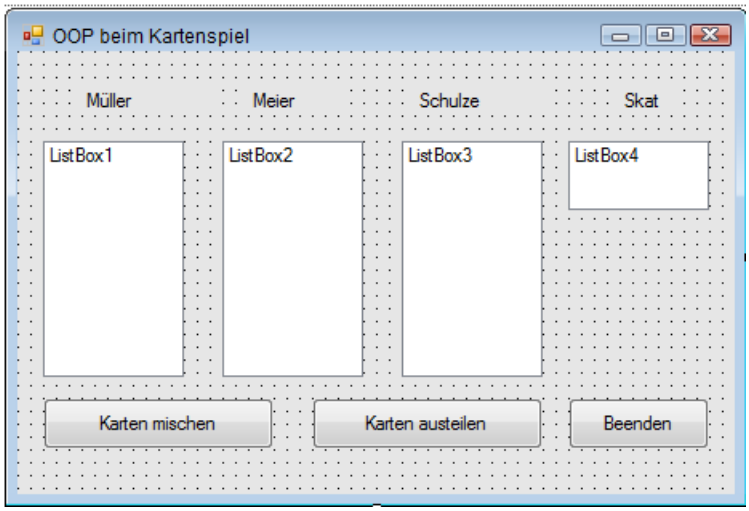
3.12.12 OOP beim Kartenspiel erlernen

Das vorliegende Beispiel soll Ihnen helfen, Ihre OOP-Grundlagen "spielend" weiter auszubauen. Es wird gezeigt, wie die in einem Array abgespeicherten *Karten*-Objekte durch einen Konstruktor erzeugt werden und selbst wiederum nach außen als Eigenschaften einer Klasse *Spiel* in Erscheinung treten.

HINWEIS: Zum Verständnis dieses Praxisbeispiels ist die Kenntnis der Skatregeln keinesfalls Voraussetzung. Es reicht aus zu wissen, dass das Spiel aus 32 Karten besteht und jeder der drei Spieler zu Beginn 10 Karten erhält und die restlichen zwei davon im so genannten "Skat" verbleiben.

Oberfläche

Die folgende Abbildung bedarf wohl keines weiteren Kommentars.



Quellcode

Es dient der Übersicht, wenn man für jede Klasse ein eigenes Klassenmodul verwendet. Diesmal aber wollen wir zeigen, dass man auch Klassen ineinander verschachteln kann. In diesem Sinne implementieren wir innerhalb des Klassencodes von *Form1* die Klassen *CKarte* und *CSpiel*:

```
Public Class Form1
    Public Class CKarte
        Public Farbe As String
        Public Wert As String
```

Der Konstruktor erzeugt ein bestimmtes Kartenobjekt, welches durch Farbe und Wert charakterisiert ist:

```
        Public Sub New(f As String, w As String)
```

```
            Farbe = f
            Wert = w
        End Sub
    End Class                       ' von CKarte
```

Die Klasse *CSpiel* kapselt 32 Instanzen der Klasse *CKarte*:

```
    Public Class CSpiel
```

Die Eigenschaft *Karten* (das ist ein Array mit 32 Karten!):

```
        Public Karten(32) As CKarte
```

Eine Hilfsmethode soll das Generieren der Karten vereinfachen:

```
        Private Sub createKarten(farbe As String, a As Integer)
            Karten(a) = New CKarte(farbe, "Sieben")
            Karten(a + 1) = New CKarte(farbe, "Acht")
            Karten(a + 2) = New CKarte(farbe, "Neun")
            Karten(a + 3) = New CKarte(farbe, "Zehn")
            Karten(a + 4) = New CKarte(farbe, "Bube")
            Karten(a + 5) = New CKarte(farbe, "Dame")
            Karten(a + 6) = New CKarte(farbe, "König")
            Karten(a + 7) = New CKarte(farbe, "As")
        End Sub
```

Der Konstruktor erzeugt und füllt das Spiel der Reihe nach mit allen 32 Karten:

```
        Public Sub New()
            Call createKarten("Eichel", 1)
            Call createKarten("Grün", 9)
            Call createKarten("Rot", 17)
            Call createKarten("Schell", 25)
        End Sub
```

Die Methode zum Mischen der Karten:

```
        Public Sub mischen()
            Dim z, i As Integer
```

Der Zwischenspeicher für den Kartentausch:

```
            Dim tmp As CKarte
            Dim rnd As New Random()
```

Alle Karten nacheinander durchlaufen:

```
            For i = 1 To Karten.Length - 1
```

Den Index einer zufälligen anderen Karte bestimmen:

```
                z = rnd.Next(1, Karten.Length)
```

Die zufällige Karte mit der aktueller Karte vertauschen:

```
                tmp = Karten(z)
                Karten(z) = Karten(i)
```

```
            Karten(i) = tmp
        Next
    End Sub
End Class                ' von CSpiel
```

HINWEIS: Aus Gründen der Einfachheit bleibt der Index 0 des *Karten*-Arrays ungenutzt.

Nun kommen wir zum eigentlichen Klassencode von *Form1*:

Ein neues Kartenspiel wird erzeugt:

```
Dim spiel As New CSpiel()
```

Eine Hilfsroutine zum Löschen der Anzeige:

```
Private Sub loeschen()
    ListBox1.Items.Clear(): ListBox2.Items.Clear()
    ListBox3.Items.Clear(): ListBox4.Items.Clear()
End Sub
```

Alle Karten mischen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    loeschen()
    spiel.mischen()
End Sub
```

Alle Karten austeilen:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
   loeschen()
```

Die Karten für Müller:

```
    For i As Integer = 1 To 10
        ListBox1.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die Karten für Meier:

```
    For i As Integer = 11 To 20
        ListBox2.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die Karten für Schulze:

```
    For i As Integer = 21 To 30
        ListBox3.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die restlichen zwei Karten wandern in den Skat:

```
    For i As Integer = 31 To 32
        ListBox4.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

```
    End Sub
    ...
End Class                       ' Form1
```

Test

Wenn Sie, vor lauter Ungeduld, unmittelbar nach Programmstart auf die Schaltfläche *Karten austeilen* klicken, werden Sie von allen drei Spielern laute Protestrufe ernten, da die Karten offensichtlich noch nicht gemischt wurden:

Erst nach ein- oder mehrmaligem Klick auf die Schaltfläche *Karten mischen* hat die Gerechtigkeit ihren Einzug gehalten und der Zufall bestimmt, welche Karten die Spieler Müller, Meier und Schulze erhalten:

Kapitel 4

Arrays, Strings, Funktionen

Ausgerüstet mit den in Kapitel 3 vermittelten OOP-Grundkenntnissen können wir nun die sprachlichen Grundlagen von Visual Basic weiter ausbauen.

Wir wollen Ihnen im vorliegenden Kapitel weiteres Handwerkszeug, wie den objektorientierten Umgang mit Arrays und Zeichenketten, Zahlenformatierungen, mathematischen und Datumsfunktionen und anderen Features 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.

Im Folgenden wollen wir zunächst die für den Einstieg wichtigsten Grundlagen vermitteln und auf die komplexeren (objektbasierten) Array-Features nur am Rand eingehen.

4.1.1 Ein Array deklarieren

Mit der *Dim*-, *Private-, Public*- oder *Static*-Anweisung kann man nicht nur normale Variablen, sondern auch ein- und mehrdimensionale Arrays definieren.

HINWEIS: Der untere Index eines Arrays ist standardmäßig null (0).

Der Datentyp des Arrays kann beliebig sein. Für ein eindimensionales Array gilt folgende Syntax:

SYNTAX: *Zugriffsmodifizierer arrName*(obererIndex) As *Datentyp*

BEISPIEL 4.1: Eine Array-Variable *a*, die 101 Gleitkommawerte aufnehmen kann, wird deklariert.

VB
```
Dim A(100) As Single
```

4.1.2 Zugriff auf Array-Elemente

Der Zugriff auf einzelne Elemente des Arrays erfolgt wie bei normalen Variablen, nur dass dem Namen der Array-Variablen der in Klammern stehende Index folgt.

BEISPIEL 4.2: (Fortsetzung) Zugriff auf das im Vorgängerbeispiel deklarierte Array

VB
```
A(0) = 12.5     ' Schreibzugriff auf das unterste Element
Dim b As Single
b = A(100)      ' Lesezugriff auf das oberste Element
```

4.1.3 Oberen Index ermitteln

Wir wissen mittlerweile, dass der untere Index eines Arrays immer null 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 zurück gibt. Der obere Index ist immer um Eins kleiner als die Länge.

BEISPIEL 4.3: Die Länge des in den Vorgängerbeispielen verwendeten Arrays wird angezeigt.

VB
```
MessageBox.Show(A.Length.ToString)          ' zeigt 101, oberer Array-Index ist 100
```

HINWEIS: Als weitere Alternativen bieten sich die *UBound*-Funktion (für die allerdings die *Microsoft.VisualBasic*-Klassenbibliothek importiert werden muss), sowie die Funktionen *GetLength(0)* und *GetUpperBound(0)* an. Letztere kommen vor allem bei mehrdimensionalen Arrays zum Einsatz (siehe 4.1.7).

4.1.4 Explizite Arraygrenzen

Bei diesem Feature handelt es sich wohl eher um Augenwischerei, denn nach wie vor beginnt in Visual Basic der unterste Index eines Arrays mit 0. Allerdings können Sie bei der Deklaration die 0 explizit hinschreiben, um den Code lesbarer zu machen.

BEISPIEL 4.4: Deklaration eines Integer-Arrays mit sechs Feldern

VB
```
Dim a(5) As Integer       ' alte Schreibweise (gilt nach wie vor!)
Dim b(0 To 5) As Integer  ' neuere Schreibweise
Dim b(1 To 5) As Integer  ' Fehler!!!
```

4.1.5 Arrays erzeugen und initialisieren

In der Regel werden Arrays bei der Erzeugung mit ihrem Leerwert (0 oder Leerstring "") initialisiert, so wie dass auch bei den normalen Datentypen der Fall ist. Um bereits beim Erzeugen Werte zuzuweisen, müssen Sie diese in geschweiften Klammern { }, getrennt durch Kommata, auflisten.

In diesem Fall müssen Sie auf die Angabe der Indexgrenze verzichten, denn diese ergibt sich ja automatisch aus der Anzahl der übergebenen Werte.

BEISPIEL 4.5: Ein *String*-Array mit fünf initialisierten Feldern

VB
```
Dim arbeitsTage() As String = {"Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag"}
```

Aber auch die Verwendung eines Collection-Initialisierers ist möglich, was den Code etwas vereinfacht.

BEISPIEL 4.6: Eine alternative Realisierung des Vorgängerbeispiels

VB
```
Dim arbeitsTage = {"Montag", "Dienstag", "Mittwoch", "Donnerstag", "Freitag"}
```

BEISPIEL 4.7: Erzeugen und initialisieren eines *Integer*-Arrays

VB
```
Dim zahlen = {0, 1, 2, 3, 4, 5, 6, 7}
```

4.1.6 Zugriff mittels Schleife

Für den sequenziellen Zugriff auf alle Array-Elemente eignen sich prinzipiell alle Schleifenanweisungen.

BEISPIEL 4.8: Ein *Integer*-Array wird mit den Zahlen 0 ... 100 gefüllt.

VB
```
Dim A(100) As Integer

For i As Byte = 0 To 100
    A(i) = i
Next i
```

For Each-Schleife

Eine weitere wichtige Zugriffsart ist die *For Each*-Schleife, mit welcher alle Elemente des Arrays durchlaufen werden können, ohne dass deren Anzahl bekannt sein müsste.

SYNTAX:
```
For Each Element In Array
  ' Anweisungen
[Exit For]
  ' Anweisungen
Next [Element]
```

BEISPIEL 4.9: Das im Vorgängerbeispiel gefüllte Array wird ausgelesen.

VB
```
For Each i As Integer In A
    MessageBox.Show(i.ToString)
Next i
```

HINWEIS: In der Regel ist das Durchlaufen einer Auflistung mit *For-Next* schneller als mit *For Each*.

Eine besonders wichtige Rolle spielt die *For Each*-Schleife im Zusammenhang mit dem Durchlaufen von Objektauflistungen (Collections).

4.1.7 Mehrdimensionale Arrays

Ein Array darf auch mehr als nur eine einzige Dimension haben. Die einzelnen Dimensionen werden bei Deklaration bzw. Zugriff durch Kommas voneinander getrennt. 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 nur eine Variable des deklarierten Datentyps aufnehmen können.

BEISPIEL 4.10: Ein (zweidimensionales) Gleitkomma-Array mit 10 Zeilen und 20 Spalten wird deklariert.

VB
```
Dim C(9, 19) As Single     ' C(0,0) ... c(9, 19)
```

Mit den folgenden Anweisungen legen Sie in jedes Fach den Wert *5.5* ab:

```
For i As Integer = 0 To 9         ' für alle Zeilen
  For j As Integer = 0 To 19        ' für alle Spalten
    C(i,j) = 5.5
  Next j
Next i
```

Mehrdimensional initialisieren

Auch mehrdimensionale Arrays lassen sich mit Anfangswerten initialisieren, die Werte der einzelnen Dimensionen werden in geschweiften Klammerpaaren hintereinander aufgeführt.

BEISPIEL 4.11: Ein zweidimensionales *String*-Array wird erzeugt und mit Namen und Wohnort von drei Personen initialisiert.

VB
```
Dim Personen(,) As String = {{"Meier", "Berlin"}, {"Schultze", "Leipzig"},
                             {"Krause", "Bonn"}}
Dim s As String = Personen(2,1)
```

Die folgende Anweisung zeigt "Bonn":

```
MessageBox.Show(s)
```

Die Inhalte des Arrays:

	0	1	2
0	Meier	Schultze	Krause
1	Berlin	Leipzig	Bonn

HINWEIS: Zur Bestimmung der Länge der einzelnen Dimensionen eines mehrdimensionalen Arrays ist die *Length*-Eigenschaft ungeeignet. Verwenden Sie stattdessen die *GetLength*- oder *GetUpperBound*-Methode.

4.1.8 Dynamische Arrays

Wenn die Größe des Arrays nicht konstant bleibt, sondern sich während der Programmausführung ändert, spricht man von einem *dynamischen* Array.

HINWEIS: Grundsätzlich sind alle mit einer .NET-Sprache deklarierten Arrays dynamisch!

ReDim-Anweisung

Man kommt nicht um das Deklarieren dynamischer Arrays herum, wenn die Array-Abmessungen erst zur Laufzeit bekannt sind. Dazu können Sie eine Array-Variable zunächst "leer" deklarieren, indem Sie die Angabe der Dimension in den ()-Klammern einfach leer lassen und erst später mittels *ReDim*-Anweisung die Array-Abmessungen festlegen.

BEISPIEL 4.12: Die Array-Größe wird erst zur Laufzeit festgelegt.

VB
```
Dim A() As Integer     ' Array-Variable "leer" deklarieren
Dim i As Byte
...
i = 100
ReDim A(i)             ' Array-Größe festlegen
```

Auch um die Dimension eines Arrays nachträglich zu ändern, verwenden Sie die *ReDim*-Anweisung.

BEISPIEL 4.13: Wir haben eine Array-Variable *a* deklariert, die 100 Gleitkommawerte aufnehmen kann, und wollen später im Programm die Dimension auf 150 erhöhen.

VB
```
Dim A(99) As Single    ' Anfangsgröße festlegen
...                    ' mit dem Array arbeiten
ReDim A(150)           ' Array vergrößern, alle Felder werden auf 0 zurückgesetzt
```

HINWEIS: Bei mehrdimensionalen Arrays können Sie nur die letzte Dimension verändern!

Bei einem Umdimensionieren wie im obigen Beispiel gehen die alten Feldinhalte verloren, einen Ausweg zeigt die *Preserve*-Option.

Preserve-Option

Die *Preserve*-Option sorgt dafür, dass durch das Umdimensionieren der alte Feldinhalt nicht zerstört wird.

BEISPIEL 4.14: Das Vorgängerbeispiel wird so geändert, dass der alte Feldinhalt erhalten bleibt.

VB
```
Dim A(99) As Single    ' Anfangsgröße festlegen
...                    ' mit dem Array arbeiten
ReDim Preserve A(150)  ' Array vergrößern, die Felder 0 ... 99 behalten ihren Inhalt
```

HINWEIS: Wesentlich schneller geht das dynamische Hinzufügen mit Instanzen der *ArrayList*-Klasse (siehe 5.3.5)!

4.1.9 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. Doch Achtung – hier ist ein wesentlicher Unterschied zu beachten:

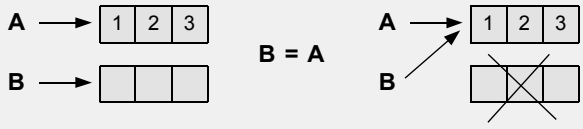
HINWEIS: Alle Arrays unter .NET sind *Referenztypen*, speichern also direkt keine Werte, sondern nur Referenzen (Zeiger) auf einen bestimmten Speicherbereich. Normale Datentypen hingegen zählen zu den *Wertetypen*.

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 4.15: Ein mit den Werten 1, 2 und 3 initialisiertes Array *A* wird einem Array *B* zugewiesen, anschließend wird im Array B ein Feld geändert.

VB
```
Dim A() As Integer = {1, 2, 3}
Dim B(2) As Integer
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 *B*-Arrays? Er steht nicht mehr zur Verfügung und wird bei nächster Gelegenheit vom Garbage Collector entsorgt.



Um ein Array wirklich in ein anderes zu kopieren, könnten Sie zunächst einen umständlichen Weg beschreiten und die Werte einzeln zuweisen.

BEISPIEL 4.16: Das Vorgängerbeispiel wird so geändert, dass ein echtes Kopieren stattfindet.

VB
```
Dim A() As Integer = {1, 2, 3}
Dim B(2) As Integer
Dim i As Byte
For i = 0 To 2
  B(i) = A(i)
Next 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 4.1.12.

4.1.10 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 schwierig zu verstehen, wenn man ein plausibles Beispiel parat hat.

BEISPIEL 4.17: Spieler einer Fußballmannschaft

VB
```
Structure Spieler
    Dim name As String
    Dim tore As Integer
    Dim gehalt As Decimal
    Dim auswahlspieler As Boolean
End Structure
```

Deklaration und Zugriff:

```
Dim AlleSpieler(10) As Spieler     ' eine Mannschaft
AlleSpieler(0).name = "Kahn"
AlleSpieler(3).name = "Klose"
```

BEISPIEL 4.18: Eine komplette Fußballmannschaft:

VB
```
Structure Mannschaft
    Dim verein As String
    Dim alleSpieler() As Spieler  ' Verweis auf Substruktur!
    Dim trainer As String
End Structure
```

Deklaration und Zugriff:

```
Dim Mannschaft1 As Mannschaft, Mannschaft2 As Mannschaft
With Mannschaft1
   ReDim .alleSpieler(10)
   .verein = "Bayern München"
```

BEISPIEL 4.18: Eine komplette Fußballmannschaft:

```
    .alleSpieler(0).name = "Lehmann"
    .alleSpieler(3).tore = 12
    .trainer.name = "Magatz"
  End With
```

BEISPIEL 4.19: Alle Spieler mit mehr als zehn Toren werden zu Auswahlspielern:

VB
```
For i = 0 To Mannschaft1.alleSpieler.Length - 1
  With Mannschaft1.alleSpieler(i)
     If .tore > 10 Then .auswahlspieler = True
  End With
Next i
```

4.1.11 Löschen von Arrays

Um alle Elemente eines Arrays zu löschen, verwenden Sie die *Erase*-Anweisung. In diesem Fall bleibt die Array-Variable erhalten.

BEISPIEL 4.20: Das Array *A* wird gelöscht und anschließend neu dimensioniert.

VB
```
Erase A
ReDim A(5, 10)
```

Eine weitere Löschalternative zeigt das folgende Beispiel (in diesem Fall wird auch die Arrayvariable zur "Entsorgung" freigegeben).

BEISPIEL 4.21: Setzen einer Array-Variablen auf *Nothing*

VB
```
A = Nothing
```

HINWEIS: Wenn Sie nur die Werte, aber nicht die Elemente des Arrays löschen wollen, können Sie auch die *Clear*-Methode der *Array*-Klasse anwenden (siehe 4.1.12).

4.1.12 Eigenschaften und Methoden von Arrays

Um das Arbeiten mit Arrays zu vereinfachen, müssen wir uns mit deren *Eigenschaften* und *Methoden* anfreunden. 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-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[1].

Wichtige Instanzeneigenschaften und -methoden

Eigenschaft bzw. Methode	Beschreibung
`Length`	... liefert Gesamtanzahl der Array-Elemente
`Rank`	... liefert die Anzahl der Array-Dimensionen
`Clone() As Array`	... kopiert alle Elemente aus Quell- in Ziel-Array
`CopyTo(arr As Array, index As Integer)`	... kopiert Elemente ab *Index* von Quell- in Ziel-Array
`GetLength(dim As Integer) As Integer`	... liefert die Anzahl Elemente einer bestimmten Dimension
`GetUpperBound(dim As Integer) As Integer`	... liefert den oberen Index einer bestimmten Dimension
`Initialize()`	... initialisiert alle Array-Elemente auf ihre Standardwerte

Array-Dimensionen bestimmen

Es gibt verschiedene Möglichkeiten, um die Anzahl von Elementen, die zu einer bestimmten Array-Dimension gehören, zu ermitteln.

BEISPIEL 4.22: Verschiedene Varianten zur Längenbestimmung eines zweidimensionalen Arrays

VB
```
Dim A(10, 12) As Single
Dim n As Integer
n = A.Length                ' 143     (Gesamtzahl der Elemente = 11 x 13 = 143)
n = A.GetLength(0)          '  11
n = A.GetLength(1)          '  13
n = A.GetUpperBound(0)      '  10
n = A.GetUpperBound(1)      '  12
n = UBound(A)               '  10
```

Arrays kopieren

Die *Clone*-Methode bietet eine Möglichkeit zum 1:1-Kopieren von Arrays. Die Abmessungen des Ziel-Arrays müssen dabei **nicht** unbedingt mit denen des Quell-Arrays übereinstimmen.

BEISPIEL 4.23: Der Inhalt des Arrays A wird in das Array B kopiert.

VB
```
Dim A() As Integer = {1, 2, 3}
Dim B() As Integer
```

[1] Die exakte Syntax entnehmen Sie bitte der Dokumentation.

BEISPIEL 4.23: Der Inhalt des Arrays A wird in das Array B kopiert.

```
B = CType(A.Clone(), Integer())      ' man achte auf die Klammern bei Integer()!
B(0) = 5
MessageBox.Show(A(0).ToString)       ' zeigt "1" an
MessageBox.Show(B(0).ToString)       ' zeigt "5" an
```

Da die *Clone*-Methode einen Wert vom *Object*-Datentyp zurückgibt, musste mit *CType* in einen *Integer* konvertiert werden.

Wichtige (statische) Methoden der Array-Klasse

Die folgenden Mitglieder der *Array*-Klasse sind statisch, können also ohne vorhandene Instanz eines Datentyps aufgerufen werden:

Eigenschaft bzw. Methode	Beschreibung
BinarySearch(*arr* As Array, *val* As Object [, *comp* As IComparer]) As Integer	... dient zur (schnellen) binären Suche in einem sortierten (!) Array
BinarySearch(*arr* As Array, *index* As Integer, *length* As Integer, *val* As Object [, *comp* As IComparer]) As Integer	... dient zur (schnellen) binären Suche ab *index* und für *length*-Elemente in einem sortierten (!) Array
Clear(*arr* As Array, *index* As Integer, *length* As Integer)	... löscht die Werte der Elemente ab *index* für *length*-Elemente (die Elemente bleiben erhalten!)
IndexOf(*arr* As Array, *val* As Object [, *start* As Integer] [*count* As Integer]) As Integer	... dient zur (langsamen) sequenziellen Suche in einem Array
Sort(*arr* As Array [, *index* As Integer, *len* As Integer] [*comp* As IComparer]) As Array	... dient zur vollständigen bzw. teilweisen Sortierung eines Arrays

Suchen in Arrays

Das folgende Beispiel bezieht sich auf die statischen Methoden *Sort* und *BinarySearch.*

BEISPIEL 4.24: Ein eindimensionales *Integer*-Array mit 10 Elementen wird nach der Zahl 11 durchsucht. Vorher wird es sortiert, um *BinarySearch* anwenden zu können.

VB

```
Dim A() As Integer = {13, 11, 5, 3, 12, 1, 3, 6, 4, 2}
Dim i As Integer
Array.Sort(A)                        ' neue Reihenfolge {1, 2, 3, 3, 4, 5, 6, 11, 12, 13}
For i = 0 To A.Length - 1
   MessageBox.Show(A(i).ToString)    ' zeigt sortierte Reihenfolge an
Next i
i = Array.BinarySearch(A, 11)        ' Suche nach der Zahl 11
If i > 0 Then
  MessageBox.Show(i.ToString)        '  gefunden an Position 7 im sortierten Array
Else
```

BEISPIEL 4.24: Ein eindimensionales *Integer*-Array mit 10 Elementen wird nach der Zahl 11 durchsucht. Vorher wird es sortiert, um *BinarySearch* anwenden zu können.

```
  MessageBox.Show("Nichts gefunden!")
End If
```

4.1.13 Übergabe von Arrays

In der Parameterliste von Funktionen/Prozeduren können auch Arrays übergeben werden. Dazu deklarieren Sie diese mit leeren Klammern.

BEISPIEL 4.25: Eine Funktion berechnet den Mittelwert eines übergebenen *Double*-Arrays.

VB

```
Public Function Mittelwert(zahlen() As Double) As Double
    Dim i As Integer, sum As Double = 0
    Dim z As Integer = zahlen.Length
    For i = 0 To z - 1
         sum += zahlen(i)
    Next i
    Return sum / z
End Function
```

Die Verwendung:

```
Dim zahlen() As Double = {1.5, 2, 3.8, 0.7}       ' Array mit vier Werten
Dim mw As Double = Mittelwert(zahlen)             ' Aufruf der Funktion
MessageBox.Show(mw.ToString)                      ' zeigt den Durchschnitt "2"
```

Funktionen sind natürlich auch zur Rückgabe von Arrays in der Lage.

BEISPIEL 4.26: Die Funktion liefert ein zweidimensionales String-Array.

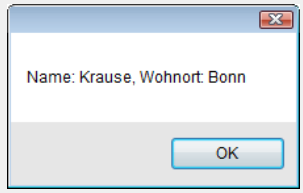
VB

```
Public Function Kundenliste() As String(,)
   Dim Kunden(,) As String =  {{"Meier", "Berlin"}, {"Schultze", "Leipzig"},{"Krause",
                               "Bonn"}, {"Schneider", "München"}}
   Return Kunden
End Function
```

Bei Verwendung der Funktion achten Sie darauf, dass der untere Feldindex immer mit null beginnt:

```
Dim liste(,) As String = Kundenliste()
MessageBox.Show("Name: " & liste(2, 0) & ", Wohnort: " & liste(2, 1))
```

Ergebnis



4.2 Zeichenkettenverarbeitung

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 Strings zuweisen

Ebenso wie Arrays sind auch Strings *Referenztypen*, d h., eine Stringvariable 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!

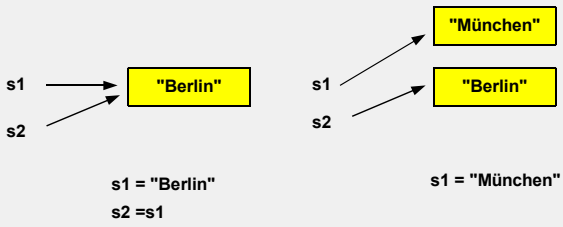
Durch diese Besonderheit wird ein Verhalten wie bei Wertetypen erreicht, obwohl es sich um Referenztypen handelt. Wer es nicht glaubt, möge folgendes Beispiel ausprobieren.

BEISPIEL 4.27: Ein String *s2* verweist auf einen String *s1*. Anschließend wird *s1* geändert.

VB

```
Dim s1 = "Berlin"
Dim 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 eines Strings

Ähnlich wie beim *Array*-Datentyp haben wir es auch hier mit zwei Gruppen von Eigenschaften/Methoden zu tun:

- Instanzeigenschaften und -methoden und
- Klasseneigenschaften und -methoden.

Während Erstere auf Objekte des *String*-Datentyps angewendet werden, sind Letztere statische Mitglieder der *String*-Klasse.

Wichtige Instanzeneigenschaften und -methoden

Eigenschaft/Methode	Beschreibung
`Chars()`	... liefert die Zeichen als *Char*-Array
`Length`	.. liefert die Anzahl der Zeichen
`EndsWith(val As String) As Boolean`	... testet, ob String mit einem bestimmten String endet
`IndexOf(val As {String|Char} [, startIndex As Integer][,count As Integer]) As Integer`	... sucht Position eines übergebenen Zeichens/Strings, liefert -1 wenn erfolglos
`Insert(startIndex As Integer, val As String) As String`	... fügt String ab *startIndex* ein
`PadLeft(totalWidth As Integer [,paddingChar As Char]) As String`	... füllt String links mit einer Anzahl Zeichen auf, bis Gesamtbreite *totalWidth* ereicht ist
`PadRight(totalWidth As Integer [,paddingChar As Char]) As String`	... wie *PadLeft*, nur rechts
`Remove(start As Integer, count As Integer) As String`	... löscht eine Anzahl von Zeichen ab einer bestimmten Position
`Replace(alt As Char|String, neu As Char|String) As String`	... ersetzt alle Zeichen bzw. Teilstrings
`Split(separator As Char, count Integer) As String()`	... zerlegt String in Teilstrings auf Basis eines Trennzeichens in ein String-Array
`StartsWidth(val As String) As Boolean`	... testet, ob String mit einem bestimmten String beginnt
`SubString(start As Integer, [length As Integer] As String`	... liefert einen Teilstring ab einer bestimmten Position (in einer bestimmten Länge)
`ToCharArray([start As Integer,] length As Integer]) As Char()`	... liefert String (oder Teilstring) als *Char*-Array
`ToLower([cult As CultureInfo]) As String`	... konvertiert alle Zeichen in Kleinbuchstaben
`ToUpper([cult As CultureInfo]) As String`	... konvertiert alle Zeichen in Großbuchstaben
`Trim([trimChars() As Char]) As String`	... löscht alle vor- oder nachlaufenden Leerzeichen (oder andere)

Einige Beispiele für Instanzeneigenschaften

Um eine einzelne *Char*-Variable aus einer *String*-Variablen herauszukopieren, greifen Sie auf die *Chars*-Eigenschaft des Strings zurück.

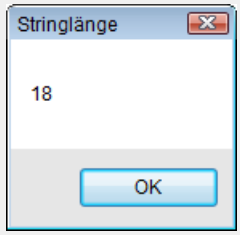
BEISPIEL 4.28: Das erste Zeichen (es hat immer den Index 0!) wird herausgefiltert.

VB
```
Dim s As String = "Hallo", c As Char
c = s.Chars(0)           ' liefert "H"
```

BEISPIEL 4.29: Die Anzahl Zeichen einer *TextBox* wird in einem Meldungsfenster angezeigt.

VB
```
TextBox1.Text = "Hallo liebe Leute!"
MessageBox.Show(TextBox1.Text.Length.ToString, "Stringlänge")
```

Ergebnis



BEISPIEL 4.30: Die Länge eines Strings wird in einer *TextBox* angezeigt.

VB
```
Dim s As String, len As Integer
s = "Wie lang ist dieser Text?"
len = s.Length
TextBox1.Text = len.ToString            ' zeigt "25"
```

Einige Beispiele für Instanzenmethoden

BEISPIEL 4.31: Alles herauskopieren ab Position 8

VB
```
Dim s As String = "Macht Visual Basic wirklich Spaß?"
TextBox1.Text = s.Substring(8)          ' zeigt "sual Basic wirklich Spaß?"
```

BEISPIEL 4.32: Sechs Zeichen ab Position 8 herauskopieren

VB
```
Dim s As String = "Macht Visual Basic wirklich Spaß?"
TextBox1.Text = s.Substring(8, 6)       ' zeigt "sual B"
```

BEISPIEL 4.33: Suchen nach einer bestimmten Zeichenfolge

VB
```
Dim s As String = "Das Wetter ist heute warm und sonnig!"
TextBox1.Text = s.IndexOf("warm").ToString      ' zeigt "21"
```

BEISPIEL 4.34: Ersetzen einer bestimmten Zeichenfolge

VB
```
Dim s As String = "Ist Visual Basic nicht eine Herausforderung?"
TextBox1.Text = s.Replace("Herausforderung", "Zumutung")
' zeigt "Ist Visual Basic nicht eine Zumutung?"
```

BEISPIEL 4.35: Mehrfaches Ersetzen

VB
```
Dim s As String = "Ist Visual Basic nicht eine Herausforderung?"
TextBox1.Text = s.Replace("e", "E")     ' zeigt "Ist Visual Basic nicht EinE
HErausfordErung?"
```

BEISPIEL 4.36: In Groß- oder Kleinbuchstaben verwandeln

VB
```
Dim s As String = "Ist Visual Basic nicht umwerfend schön?"

TextBox1.Text = s.ToUpper     ' zeigt "IST VISUAL BASIC NICHT UMWERFEND SCHÖN?"
TextBox1.Text = s.ToLower     ' zeigt "ist visual basic nicht umwerfend schön?"
```

4.2.3 Kopieren eines Strings in ein Char-Array

BEISPIEL 4.37: *String* in *Char*-Array

VB
```
Dim s As String = "Hallo"
Dim c As Char() = s.ToCharArray()
Dim i As Integer

For i = 0 To c.GetUpperBound(0)
  MessageBox.Show(c(i).ToString)    ' zeigt nacheinander "H", "a", "l", "l", "o"
Next i
```

Obwohl es eigentlich mit den Stringmethoden direkt nichts zu tun hat, soll der Vollständigkeit halber auch der umgekehrte Weg gezeigt werden.

BEISPIEL 4.38: *Char*-Array in *String*

VB
```
Dim c As Char() = {"H"c, "a"c, "l"c, "l"c, "o"c}
Dim s As String = c          ' direkte Zuweisung möglich!
MessageBox.Show(s)           ' zeigt "Hallo"
```

4.2.4 Wichtige (statische) Methoden der String-Klasse

Die folgenden Mitglieder der *String*-Klasse sind statisch, können also ohne vorhandene Instanz eines Datentyps aufgerufen werden:

Methode	Beschreibung
Compare(*str1* As String, *str2* As String [*ignoreCase* As Boolean]) As Integer	... vergleicht zwei Strings mit (oder ohne) Berücksichtigung der Groß-/Kleinschreibung
Copy(*str* As String) As String	... erzeugt neue Stringinstanz
Empty()	... liefert einen Leerstring
Format(*format* As String, *arg0* As Object [, *arg1* As Object] [, *arg2* As Object]) As String	... formatiert einen String aufgrund von Formatierungsargumenten
Join(*separator* As String, *val*() As String [*start* As Integer, *count* As Integer]) As String	... erzeugt aus String-Array einen einzelnen String, Teile werden durch Separator getrennt

Stringvergleich

BEISPIEL 4.39: Vergleich des Inhalts zweier TextBoxen unter Berücksichtigung der Groß-/Kleinschreibung

VB

```
Dim i As Integer            ' Rückgabewert der Compare-Methode
i = String.Compare(TextBox1.Text, TextBox2.Text, False)
Select Case i
   Case -1 : Label1.Text = "String 1 ist kleiner als String 2 !"
   Case 0 : Label1.Text = "String 1 ist gleich String 2 !"
   Case 1 : Label1.Text = "String 1 ist größer als String 2 !"
End Select
```

Ergebnis

Strings zerlegen und wieder zusammenbauen

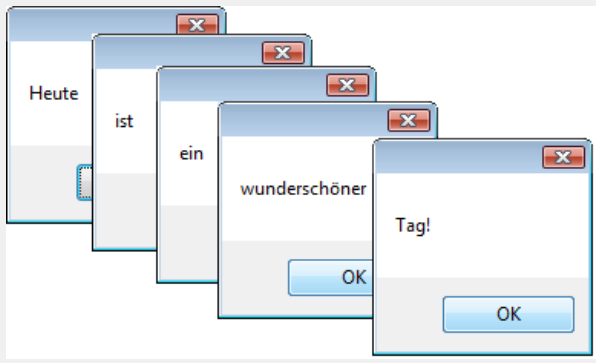
Ein gutes Beispiel für das Zusammenwirken von Instanzen- und Klassenmethoden ist das Zerlegen und Zusammenfügen von Zeichenketten. Zum Einsatz kommen die *Split*-Methode (eine Instanzenmethode) und die *Join*-Methode (eine statische Methode der *String*-Klasse).

BEISPIEL 4.40: Ein *String* wird in Teilstrings zerlegt.

VB

```
Dim A() As String
Dim s As String = "Heute ist ein wunderschöner Tag!"
A = s.Split(" "c)              ' Trennzeichen ist das Leerzeichen!
Dim i As Integer
For i = 0 To A.GetUpperBound(0)
     MessageBox.Show(A(i))      ' zeigt Array-Elemente nacheinander an
Next i
```

Ergebnis

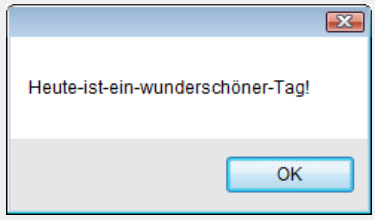


BEISPIEL 4.41: Der *String* wird wieder zusammengefügt, neues Trennzeichen ist "-":

VB

```
s = String.Join("-", A)
MessageBox.Show(s)
```

Ergebnis



BEISPIEL 4.42: Der Inhalt einer mehrzeiligen *TextBox* (*MultiLine*-Eigenschaft ist *True*) wird zeilenweise in einem Array abgelegt und anschließend wieder zurück in die *TextBox* kopiert.

VB

```
Dim A() As String
A = TextBox1.Text.Split(CChar(vbCrLf))          ' zeilenweise ab in's Array!
...
TextBox1.Text = String.Join(CChar(vbCrLf), A)   ' marsch zurück in die Anzeige!
```

HINWEIS: Um die Konstante *vbCrLf* (Zeilenrücklauf und neue Zeile) nutzen zu können, müssen Sie die *Microsoft.VisualBasic*-Klassenbibliothek referenzieren (standardmäßig sollte dies bereits der Fall sein).

4.2.5 Die StringBuilder-Klasse

Dass Strings im Speicher als unveränderliche Zeichenketten abgelegt werden, hat natürlich seinen Preis. Wird der Inhalt einer String-Variablen sehr häufig geändert, so kann dies zu einer deutlichen Verringerung der Performance der Anwendung führen. In solchen Fällen sollten Sie auf die Fähigkeiten der *StringBuilder*-Klasse aus dem Namespace *System.Text* zurückgreifen.

BEISPIEL 4.43: Zu einer Stringvariablen wird zwanzigtausendmal ein Wert hinzuaddiert.

VB

Zunächst die Addition auf herkömmliche Weise:

```
Dim s As String = String.Empty
For i As Integer = 0 To 20000
  s &= i.ToString
Next i
```

Während obige Schleife mit einem modernen PC ca. 5 Sekunden Zeit beansprucht, kann das gleiche Problem mit einem *StringBuilder* in Bruchteilen einer Sekunde erledigt werden:

```
Dim sb As New System.Text.StringBuilder
For i As Integer = 0 To 20000
    sb.Append(i.ToString)
Next i
Dim s As String = sb.ToString
```

Ein ausführliches Testbeispiel zur Stringaddition mit eingebauter Zeitmessung finden Sie in

► 4.7.2 Zeichenketten mittels StringBuilder addieren

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*).

Erzeugen

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 4.44: Ein *StringBuilder*-Objekt für die Zeichenfolge "Heute ist Mittwoch!" wird erzeugt

VB
```
Dim strB As 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.

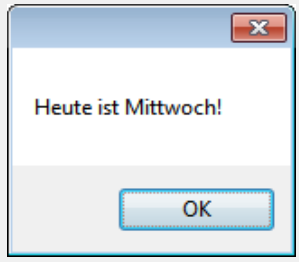
Wollen Sie die im *StringBuilder*-Objekt enthaltene Zeichenfolge auslesen, so beachten Sie:

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 4.45: Der Inhalt des im Vorgängerbeispiel erzeugten *StringBuilder*-Objekts wird ausgelesen

VB
```
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)

Eigenschaft	Beschreibung
`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 4.46: Ein *StringBuilder*-Objekt wird erzeugt, dann werden diverse Eigenschaften ausgelesen

VB
```
Dim strB As New StringBuilder("Heute ist Mittwoch!", 50)
Dim cap As Integer = strB.Capacity      '  50
Dim lg As Integer = strB.Length         '  19
Dim max As Integer = strB.MaxCapacity   '  2147483647
```

BEISPIEL 4.47: Aus der Zeichenfolge "Heute ist Mittwoch" wird das Zeichen an der 11.Position geliefert

VB
```
Dim strB As New StringBuilder("Heute ist Mittwoch")
Dim c As Char = 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 4.48: Eine Zeichenkette wird einem *StringBuilder*-Objekt zugewiesen

VB
```
Dim strB As New StringBuilder()
strB = strB.Append("Heute ist Mittwoch!")
Label1.Text = strB.ToString               ' "Heute ist Mittwoch!"
```

BEISPIEL 4.49: In die Zeichenfolge "Heute!" wird ab Position 6 die Zeichenfolge "ist Mittwoch" eingefügt

VB
```
Dim strB AS New StringBuilder("Heute !")
strB = strB.Insert(6, "ist Mittwoch")
Label1.Text = strB.ToString                ' "Heute ist Mittwoch!"
```

BEISPIEL 4.50: In der Zeichenfolge "Heute ist Mittwoch!" wird das letzte Zeichen gelöscht

VB
```
Dim strB As New StringBuilder("Heute ist Mittwoch!")
strB.Remove(strB.Length-1,1)
Label1.Text = strB.ToString                 ' "Heute ist Mittwoch"
```

BEISPIEL 4.51: In der Zeichenfolge "Heute ist Mittwoch!" wird der Wochentag ersetzt

VB
```
Dim strB As 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 braucht man reguläre Ausdrücke?

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

4.3.3 Wichtige Methoden 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

[1] SPAM-Schleudern generieren so systematisch E-Mail-Adressen (den Teil vor dem @).

Die Match-Methode

Für die Überprüfung regulärer Ausdrücke können wir die statische *Match*-Methode der *RegEx*-Klasse verwenden:

BEISPIEL 4.52: Der reguläre Ausdruck "Walth?er" wird getestet

VB
```
...
Imports System.Text.RegularExpressions
...
Dim rgEx As String = "Walth?er"
Dim testStr As String = "Waalther"            ' mit verschiedenen Strings testen
If Regex.Match(testStr, rgEx).Success Then
    MessageBox.Show("Treffer!")
Else
    MessageBox.Show("Kein Treffer!")
End If
```

Eigenschaften des Match-Objekts

Das von der *Match*-Methode gelieferte *Match*-Objekt kapselt Eigenschaften für den aktuellen Wert, dessen Index und Länge.

BEISPIEL 4.53: Eigenschaften des Match-Objekts ermitteln

VB
```
Dim m As Match = 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 4.54: NextMatch

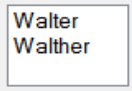
```
VB  Dim m1 As Match = Regex.Match("Walter oder Walther", "Walth?er")
    Dim m2 As Match = m1.NextMatch()
    Label1.Text = m1.ToString          ' Walter
    Label2.Text = m2.ToString          ' Walther
```

Die Matches-Methode

Die *Matches*-Methode liefert alle Treffer als Array zurück.

BEISPIEL 4.55: Eine erweiterte Version des Vorgängerbeispiels

```
VB  For Each m As Match In Regex.Matches("Walter oder Walther", "Walth?er")
        ListBox1.Items.Add(m.ToString)
    Next
```



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 4.56: Der reguläre Ausdruck "Wal(t|th)?er" wird getestet

```
VB  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 4.57: Eine kompilierte Variante des Vorgängerbeispiels

```
VB  Dim r As New Regex("Wal(t|th)?er")
```

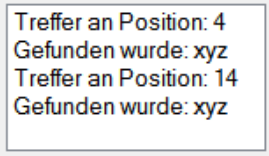
BEISPIEL 4.57: Eine kompilierte Variante des Vorgängerbeispiels

```
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 4.58: Alle Vorkommen von "xyz" in einer Zeichenkette werden ermittelt und angezeigt

VB
```
Dim rx As New Regex("xyz")
Dim mcoll As MatchCollection = rx.Matches("0123xyz4567uvwxyza")
For i As Integer = 0 To mcoll.Count - 1
     ListBox1.Items.Add("Treffer an Position: " & mcoll(i).Index)
     ListBox1.Items.Add("Gefunden wurde: " & mcoll(i).Value)
Next i
```



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 4.59: Suche ohne Berücksichtigung der Groß-/Kleinschreibung

VB
```
Dim m As Match = Regex.Match ("walter", "Walth?er", RegexOptions.IgnoreCase)
Label1.Text = m.ToString                                    ' zeigt "walter"
```

Die meisten der *RegexOptions* können innerhalb eines regulären Ausdrucks mittels eines einzelnen Zeichencodes aktiviert werden, damit lässt sich etwas Schreibarbeit sparen.

BEISPIEL 4.60: Verkürzte Notation des Vorgängerbeispiels mittels des Zeichencodes (?i)

VB
```
Dim m As Match = Regex.Match ("walter", "(?i)Walth?er")
Label1.Text = m.ToString              '  zeigt "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 4.61: Anwendung der Option *IgnorePatternWhitespace* (?x)

VB
```
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 4.62: Der reguläre Ausdruck "1+1" richtig und falsch geschrieben

VB
```
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 4.63: Die Methoden *Escape* und *Unescape*

VB
```
...
Label1.Text = Regex.Escape("1+1=2")              ' "1\+1=2"
Label2.Text = Regex.Unescape ("1\+1=2")          ' "1+1=2"
```

4.3.7 Zeichenmengen (Character Sets)

Character Sets 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
\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 4.64: Der gesuchte String muss 2 Zeichen lang sein und mit "W" beginnen, danach kann ein beliebiges Zeichen außer "a" und "e" folgen

VB
```
Dim m As Match m Regex.Match("Walter,Werner,Wolfgang", "W[^ae]")
Dim i As Integer = m.Index                  ' 14
Dim s As String = m.Value                   ' "Wo"
```

Um ein einzelnes Zeichen aus einer Zeichenmenge zu treffen, muss die Zeichenmenge in eckige Klammern eingeschlossen werden:

BEISPIEL 4.65: Wieviel Strings "Walter" oder "walter" sind enthalten?

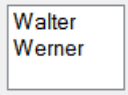
VB
```
Dim I As Integer = Regex.Matches ("Walter,wilhelm,walter.", "[Ww]alter").Count     ' 2
```

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 4.66: Vornamen, die mit "W" beginnen, mit "er" enden und in der Mitte drei beliebige Zeichen haben

VB
```
For Each m As Match In Regex.Matches("Walter,Werner,Wolfgang", "W...er"))
   ListBox1.Items.Add(m.ToString)
Next
```



Einen Zeichenbereich gibt man mittels Bindestrich an.

BEISPIEL 4.67: Die Gültigkeit eines Schachzugs überprüfen[1]

VB
```
If Regex.Match ("c2-c4", "[a-h]\d-[a-h]\d").Success Then
        MessageBox.Show("Der Zug ist gültig!")
End If
```

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 4.68: Es wird überprüft, ob der Dateiname *bild5.jpg* zu der Serie *bild0.jpg, bild1.jpg, ...* gehört

VB
```
Dim suc As Boolean = Regex.Match ("bild5.jpg", "bild\d*\.jpg").Succes    ' True
```

[1] Dieses einfache Pattern entspricht nur bedingt den Schachregeln, aber uns geht es ja hier nur um das Prinzip.

HINWEIS: Beachten Sie den Backslash \ vor dem Punkt in der Datei-Extension, er verhindert die Interpretation des Punkts als Metazeichen!

BEISPIEL 4.69: Es werden alle mit *bild...* beginnenden *.jpg*-Dateien erfasst

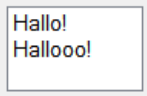
VB
```
Dim suc As Boolean = Regex.Match ("bildUrlaub_12.jpg", "bild.*\.jpg").Success    ' True
```

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 4.70: Alle "Hallo...!" mit einem oder mehreren "o" werden herausgefiltert

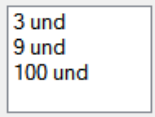
VB
```
For Each m As Match In Regex.Matches("Hallo! Hallo Hallooo!", "Hallo+!")
                ListBox1.Items.Add(m.ToString)
Next
```



Der {} Quantifizierer legt eine Anzahl oder einen Bereich für die Wiederholungen fest.

BEISPIEL 4.71: Es werden die Strings herausgefiltert, die aus einer ein- oder dreistelligen Zahl bestehen, gefolgt von " und "

VB
```
For Each m As Match In Regex.Matches("3 und 9 und 100 und 12 ergibt 124", ("\d{1,3} und"))
    ListBox1.Items.Add(m.ToString)
Next
```



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.

[1] "Behauptungen der Breite null"

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[1]. 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

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 4.72: Alle leeren Zeilen in einem mehrzeiligen Text werden erfasst

VB
```
Dim emptLines As MatchCollection = Regex.Matches (txt, "^(?=\r?$)", RegexOptions.Multiline)
' oder
Dim emptLines As MatchCollection = Regex.Matches (txt, "(?m)^(?=\r?$)")
```

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.

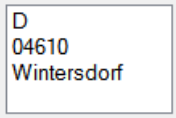
Der Wortanker *\b* wird häufig benutzt um ganze Wörter zu herauszufiltern.

BEISPIEL 4.73: Alle Wörter werden aus dem String extrahiert

VB
```
For Each m As Match In Regex.Matches("D-04610 Wintersdorf", "\b\w+\b"))
      ListBox1.Items.Add(m.Value)
Next
```

[1] Eine andere Bedeutung von ^ ist die eines Negationsoperators für Zeichenmengen (siehe Abschnitt 4 3.7).

BEISPIEL 4.73: Alle Wörter werden aus dem String extrahiert



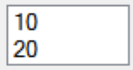
Das Pattern \w+ innerhalb der Wortbegrenzer \b ist die Kombination einer Zeichenmenge (\w) mit dem Existenzquantor + und bedeutet die mindestens einmalige Wiederholung eines Wortzeichens.

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 4.74: Das Pattern "\s(?=EUR)" steht für ein Leerzeichen, dem der String "EUR" folgt.

VB
```
Dim rx As New Regex("\d+\s(?=EUR)")
Dim mcoll As MatchCollection = rx.Matches("Heute 10 EUR, morgen 15EUR, übermorgen 20 EUR")
For i As Integer = 0 To mcoll.Count -1
       ListBox1.Items.Add(mcoll(i).Value)
Next i
```



Obwohl als Suchkriterium angegeben, gehört der String "EUR" im obigen Beispiel nicht zur gematchten Zeichenkette!

Das folgende Beispiel hat durchaus praktische Relevanz:

BEISPIEL 4.75: Ein Passwort soll mindestens 8 Zeichen enthalten, davon mindestens eine Ziffer

VB
```
Dim password As String = "geheim457"
Dim ok As Boolean = 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].

[1] Finden Sie das Warum selbst heraus!

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 4.76: Suche nach dem Wort "Aufgaben", wenn danach nicht "fleißig" oder "Fleiß" vorkommen

VB
```
Dim rex As String = "Aufgaben(?!.*(fleißig|Fleiß))"
Dim text As String = "Die übertragenen Aufgaben erfüllte er mit viel Fleiß."
Dim m As Match = Regex.Match (text, rex)
If m.Success Then
     MessageBox.Show("Bewerber ist geeignet!", "Treffer", MessageBoxButtons.OK,
                     MessageBoxIcon.Exclamation)
Else
     MessageBox.Show("Bewerber ist ungeeignet!", "Kein Treffer", MessageBoxButtons.OK,
                     MessageBoxIcon.Error)
End If
```

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 4.77: Es wird nach "gut" gesucht, zuvor darf aber nicht "noch..." stehen.

VB
```
Dim rex As String = "(?i)(?<!noch.*)gut"
Dim Text As String = "Die Leistung wird noch als Gut eingeschätzt..."

Dim m As Match = Regex.Match (text, rex)
Dim b As Boolean = 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 "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 297) 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 4.78: Wir wollen eine deutsche Postleitzahl in zwei Gruppen aufteilen: PLZ und Ortsnamen

VB
```
Dim m As Match = 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 4.79: Suche nach allen Wörtern, die mit denselben Buchstaben beginnen und enden. Der Ausdruck \1 bezieht sich auf das aktuelle Wortzeichen der ersten Gruppe (\w).

VB
```
For Each m AS Match In Regex.Matches("otto Otto ende NAMEN", "\b(\w)\w+\1\b"))
   ListBox1.Items.Add(m.Value)
Next
```

4.3.11 Text ersetzen

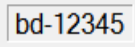
Die statische Methode *RegEx.Replace* ähnelt der Ihnen bereits bekannten Instanzenmethode *Replace* einer herkömmlichen Stringvariablen, allerdings verwenden wir jetzt als Suchkriterium einen regulären Ausdruck.

BEISPIEL 4.80: Die Großbuchstaben eines Strings werden entfernt

VB
```
Dim input As String = "AbCdE-12345"
Dim find As String = "[A-Z]"
```

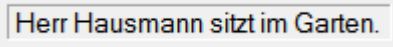
BEISPIEL 4.80: Die Großbuchstaben eines Strings werden entfernt

```
Dim replace As String = ""
Label1.Text = Regex.Replace(input, find, replace)
```



BEISPIEL 4.81: Ersetzen von "Haus" durch "Garten"

VB
```
Label1.Text = Regex.Replace("Herr Hausmann sitzt im Haus.", "\bHaus\b", "Garten")
```

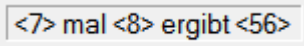


Weil wir im obigen Beispiel die Suche auf ganze Wörter begrenzen (Wortanker \b), erhalten wir nicht das Ergebnis "*Herr Gartenmann sitzt im Garten.*".

Der Ersetzungsstring kann sich mittels der Substitution $0 auf den Original-Treffer beziehen.

BEISPIEL 4.82: Alle Zahlen innerhalb eines Strings werden in spitze Klammern eingeschlossen.

VB
```
Label1.Text = Regex.Replace("7 mal 8 ergibt 56", "\d+", "<$0>")
```



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 4.83: Splitten einer Zeichenkette. Trennzeichen ist irgendeine Ziffer.

VB
```
For Each s As String In Regex.Split ("abc36b5cd", "\d"))
        ListBox1.Items.Add(s)
Next
```

Ergebnis

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 298) einschließen.

BEISPIEL 4.84: Trennelemente sind die Großbuchstaben. Diese aber sollen im Ergebnis mit erscheinen.

VB
```
Dim rex As String = "(?=[A-Z])"
Dim text As String = "JanuarFebruar März"
For Each s As String In Regex.Split(text, rex)
       ListBox1.Items.Add(s)
Next
```

Ergebnis

Weitere Erfahrungen mit regulären Ausdrücken sammeln Sie im Praxisbeispiel

► 4.7.3 Reguläre Ausdrücke testen

4.4 Datums- und Zeitberechnungen

Dieser Datentyp wird benutzt, um ein Datum einschließlich einer Uhrzeit in einer Variablen zu speichern.

4.4.1 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 4.85: Eine *DateTime*-Variable für den *25.August 2015 12:15* erzeugen und auswerten

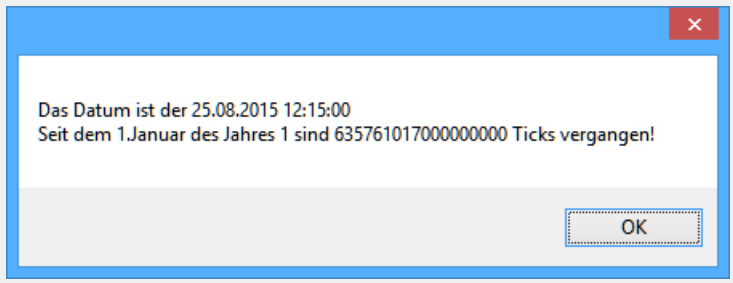
VB
```
Dim dat1 As New DateTime(2015, 8, 25, 12, 15, 0)
Dim ts As Long = dat1.Ticks
Dim s As String = "Das Datum ist der " & dat1.ToString & vbCrLf &
s &= "Seit dem 1.Januar des Jahres 1 sind " & ts.ToString & " Ticks vergangen!"
MessageBox.Show(s)
```

[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.

BEISPIEL 4.85: Eine *DateTime*-Variable für den *25.August 2015 12:15* erzeugen und auswerten

Ergebnis



Um aus den ermittelten Datum und Uhrzeit zu rekonstruieren, verwenden wir einfach einen anderen Konstruktor:

BEISPIEL 4.86: Fortsetzung des Vorgängerbeispiels

VB

```
Dim dat2 As New DateTime(ts)
MessageBox.Show(dat2.ToString)          ' "25.08.2015 12:15:00"
```

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 4.87: Die Eigenschaften eines Datums, welches in eine *TextBox* eingegeben wurde

```
Dim dat As DateTime = Convert.ToDateTime(TextBox1.Text)
Dim h As Integer = dat.Hour               ' h erhält den Wert 17
Dim m As Integer = dat.Minute             ' m erhält den Wert 3
Dim y As Integer = dat.Year               ' y erhält den Wert 1952
```

BEISPIEL 4.88: Der Wochentag für das Datum 20.8.2015 wird ermittelt

```
VB  Dim dat As DateTime = Convert.ToDateTime("20.8.2015")
    Dim day As DayOfWeek = 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.

BEISPIEL 4.89: Der wievielte Tag im Jahr ist der 20.8.2015?

```
VB  Dim dat As DateTime = Convert.ToDateTime("20.8.2015")
    Dim i As Integer = dat.DayOfYear                ' i erhält den Wert 236
```

4.4.3 Wichtige Methoden von DateTime-Variablen

Methode	Beschreibung
AddDays(*val As Double*) *As DateTime*	... 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(*val As Double*) *As DateTime*	
AddMinutes(*val As Double*)*As DateTime*	
AddMonths(*val As Double*) *As DateTime*	
AddYears(*val As Double*) *As DateTime*	
ToLongDateString() *As String*	... liefern das Datum als kurzen oder langen Datums-/Zeitstring und können als Alternative zur *ToString*-Methode eingesetzt werden.
ToLongTimeString() *As String*	
ToShortDateString() *As String*	
ToShortTimeString() *As String*	

BEISPIEL 4.90: Das Datum, was genau 2,5 Tage vor dem 23.8.2015, 16:30, liegt

```
VB  Dim dat As DateTime = Convert.ToDateTime("23.8.2015 16:30")
    dat = dat.AddDays(-2.5)
    MessageBox.Show(dat.ToString)       ' zeigt "21.08.2015 04:30:00"
```

BEISPIEL 4.91: Formatierte Ausgabe der Zeit

```
VB  Dim dat As DateTime = Convert.ToDateTime("23.8.2015 16:30")
    MessageBox.Show(dat.ToShortTimeString)              ' zeigt "16:30"
```

HINWEIS: Zahlreiche weitere Möglichkeiten der Formatierung von *DateTime*-Variablen finden Sie im Abschnitt 4.6 "Datumsformatierungen".

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(year As Integer, month As Integer) As Integer`	... liefert die Anzahl von Tagen eines Monats
`IsLeapYear(year As Integer) As Boolean`	... stellt fest, ob das Jahr ein Schaltjahr ist
`Parse(s As String) As DateTime`	... dient zum Umwandeln eines Strings in einen *DateTime*-Wert mit optionalen Kulturinformationen

BEISPIEL 4.92: Die Tage des Monats Februar 2016

VB

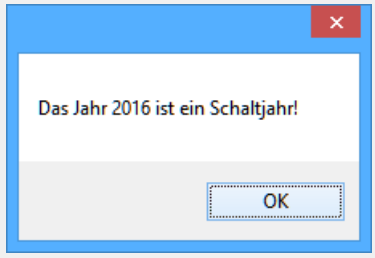
```
Dim i As Integer = DateTime.DaysInMonth(2016, 2)          ' i erhält den Wert 29
```

BEISPIEL 4.93: Ist das Jahr 2016 ein Schaltjahr?

VB

```
Dim year As Integer = 2016
Dim s As String = "Das Jahr " & year.ToString & " ist kein Schaltjahr!"
If DateTime.IsLeapYear(year) Then  s = s.Replace("kein", "ein")
MessageBox.Show(s)
```

Ergebnis

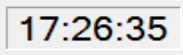


BEISPIEL 4.94: Eine Uhr anzeigen

VB

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 Sub Timer1_Tick(sender As Object, e As EventArgs)
    Label1.Text = DateTime.Now.ToString("HH:mm:ss")
End Sub
```



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 4.95: Datum und Zeit werden in jeweils eine *TextBox* eingegeben, in einen *DateTime*-Typ konvertiert und in einem *Label* angezeigt

VB
```
Dim d As DateTime = 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 4.96: Eine äquivalente Realisierung des Vorgängerbeispiels

VB
```
Dim d As DateTime = 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 4.97: Die Zeitdauer zwischen zwei Zeitpunkten wird berechnet und in Tagen, Stunden, Minuten und Sekunden aufgeschlüsselt

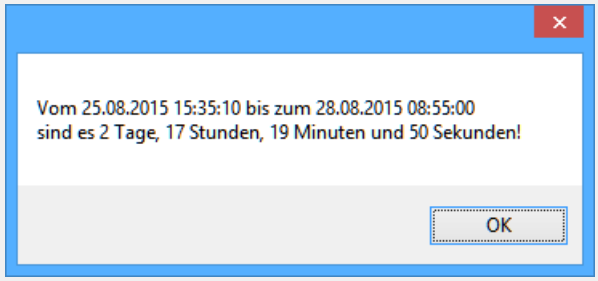
VB
```
Dim d1 As New DateTime(2015, 8, 25, 15, 35, 10)    ' 25.8.2015 15:35:10
Dim d2 As New DateTime(2015, 8, 28, 8, 55, 0)      ' 28.8.2015 08:55:00

Dim ts As New TimeSpan(d2.Ticks - d1.Ticks)

Dim tg As String = ts.Days.ToString
Dim std As String = ts.Hours.ToString
Dim min As String = ts.Minutes.ToString
Dim sek As String = ts.Seconds.ToString
```

[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).

BEISPIEL 4.97: Die Zeitdauer zwischen zwei Zeitpunkten wird berechnet und in Tagen, Stunden, Minuten und Sekunden aufgeschlüsselt

```
MessageBox.Show("Vom " & d1.ToString & " bis zum " & d2.ToString & vbCrLf &
            " sind 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.

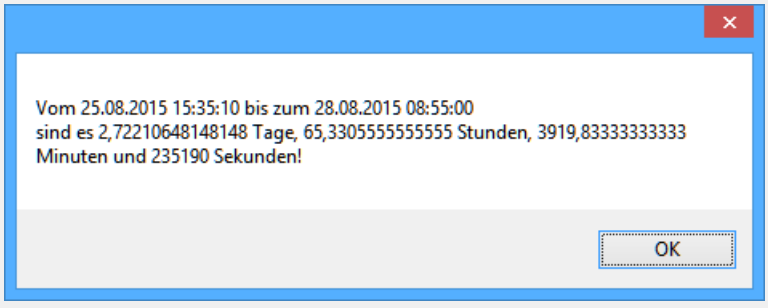
Andere Eigenschaften, die mit *Total...* beginnen, erlauben hingegen eine Auswertung der Gesamtstunden, -minuten etc. zwischen zwei Zeitpunkten.

BEISPIEL 4.98: Modifiziertes Vorgängerbeispiel mit ausgetauschten Eigenschaften (Days => TotalDays, Hours => TotalHours usw)

VB

```
  ...
  Dim tg As String = ts.TotalDays.ToString
  Dim std As String = ts.TotalHours.ToString
  Dim min As String = ts.TotalMinutes.ToString
  Dim sek As String = ts.TotalSeconds.ToString
...
```

Ergebnis



Eine nützliche Anwendung von *TimeSpan* zur Zeitmessung im Millisekundenbereich finden Sie im Praxisbeispiel

► 4.7.2 Zeichenketten mittels StringBuilder addieren

4.5 Vordefinierten Funktionen

Es folgt ein Überblick über die in der *Math*- und der *DateTime*-Klasse bereitgestellten Eigenschaften und Methoden zur Ausführung von in der Praxis häufig vorkommenden Berechnungen.

4.5.1 Mathematik

Die *Math*-Klasse stellt eine Vielzahl von (statischen) Methoden zur Verfügung, mit denen Sie die üblichen mathematischen Funktionen aufrufen können:

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)
`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. Bei *Option Strict On* (eingeschaltete strenge Typprüfung) sind deshalb die entsprechenden Konvertierungsfunktionen zu bemühen.

Eine Anwendung der *CSng*-Konvertierungsfunktion zeigt das folgende Beispiel.

Zahlen runden

BEISPIEL 4.99: Rundung einer *Single*-Zahl auf zwei Nachkommastellen

VB
```
Dim t As Single
t = CSng(Math.Round(12.477, 2))
MessageBox.Show(t.ToString)          ' zeigt "12,48"
```

Winkel umrechnen

Der Winkel wird grundsätzlich im Bogenmaß (*Radian*) angegeben. Die Umrechnungsformel lautet:

Rad = Grad * Pi /180

BEISPIEL 4.100: Der Sinus von 30 Grad

```
VB  Dim d As Double = Math.Sin(30 * Math.PI / 180)          ' liefert 0,5
```

Wurzel- und Potenzoperationen

Für diese Operationen stehen sowohl die *Pow*-Methode als auch der einfache Potenzoperator (^) zur Verfügung.

BEISPIEL 4.101: Die dritte Wurzel aus 10

```
VB  Dim d As Double = 10
    d = Math.Pow(10, 1 / 3)
    MessageBox.Show(r.ToString("0.0000"))                ' zeigt 2,1544
```

oder einfach mit dem Potenzoperator

```
    d = r ^ (1 / 3)
```

Logarithmus und Exponentialfunktionen

Log(x) und *Exp(x)* sind zueinander Umkehrfunktionen. Sie beziehen ihre Basis auf die Zahl *e* (2.718282...). Es gilt der Zusammenhang:

Exp(Log(x)) = Log(Exp(x)) = x

BEISPIEL 4.102: Der "Beweis" für obige Formel[1]

```
VB  Const x As Double = 25
    Dim y As Double
    y = Math.Exp(Math.Log(x))
    MessageBox.Show(y.ToString)    ' zeigt "25"
    y = Math.Log(Math.Exp(x))
    MessageBox.Show(y.ToString)    ' zeigt "25"
```

BEISPIEL 4.103: Nachbildung des dekadischen Logarithmus (zur Basis 10) durch eine selbst definierte Funktion

```
VB  Function Log10(x As Double) As Double
        Log10 = Math.Log(x) / Math.Log(10)
    End Function
```

[1] Den Mathematikern unter den Lesern werden zwar die Haare zu Berge stehen, aber hier geht es nur um Programmierpraxis.

BEISPIEL 4.103: Nachbildung des dekadischen Logarithmus (zur Basis 10) durch eine selbst definierte Funktion

```
...
Dim d As Double = Log10(100)
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 4.104: Das Vorgängerbeispiel wird vereinfacht.

VB
```
Dim d As Double = Math.Log10(100)
MessageBox.Show(d.ToString)                ' zeigt "2"
```

4.5.2 Datums- und Zeitfunktionen

Wir wollen wir uns hier den zahlreichen Eigenschaften und Methoden zuwenden, mit denen Datums-/Zeitberechnungen möglich sind.

Wie bereits bei der *Array*- und der *String*-Klasse gezeigt, müssen wir auch hier unterscheiden, ob die Eigenschaften/Methoden auf eine *DateTime*-Variable (Instanz) oder direkt auf die *DateTime*-Klasse angewendet werden.

Wir wollen wir uns hier den zahlreichen Eigenschaften und Methoden zuwenden, mit denen Datums-/Zeitberechnungen möglich sind.

Wie bereits bei der *Array*- und der *String*-Klasse gezeigt, müssen wir auch hier unterscheiden, ob die Eigenschaften/Methoden auf eine *DateTime*-Variable (Instanz) oder direkt auf die *DateTime*-Klasse angewendet werden.

Wichtige Instanzeneigenschaften und -methoden

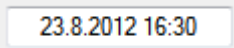
Die folgende Tabelle zeigt einen Einblick in das reichhaltige Sortiment:

Eigenschaft/Methode	Beschreibung
`Second, Minute, Hour, Day, Month, Year`	... liefern die Bestandteile eines Datums als *Integer*-Zahl
`DayOfWeek`	... liefert den Wochentag als Teil der *DayOfWeek*-Enumeration (*Monday, Tuesday,Wednesday, Thursday, Friday, Saturday, Sunday*)
`DayOfYear`	... liefert die auf das Jahr bezogene Nummer des Tages als *Integer*-Zahl

Eigenschaft/Methode	Beschreibung
`AddDays(val As Double) As DateTime`	... 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(val As Double) As DateTime`	
`AddMinutes(val As Double) As DateTime`	
`AddMonths(val As Double) As DateTime`	
`AddYears(val As Double) As DateTime`	
`ToLongDateString() As String`	... liefern das Datum als kurzen oder langen Datums- bzw. Zeitstring und können als Alternative zur *ToString*-Methode eingesetzt werden
`ToLongTimeString() As String`	
`ToShortDateString() As String`	
`ToShortTimeString() As String`	

BEISPIEL 4.105: Die Eigenschaften eines Datums, welches in eine *TextBox* eingegeben wurde

VB




```
Dim dat As Date = CDate(TextBox1.Text)
Dim h As Integer = dat.Hour                ' h erhält den Wert 16
Dim m As Integer = dat.Minute              ' m erhält den Wert 30
Dim y As Integer = dat.Year                ' y erhält den Wert 2012
```

BEISPIEL 4.106: Der Wochentag für das Datum 23.8.2012 wird ermittelt.

VB

```
Dim dat As Date = #8/23/2012#
Dim day As DayOfWeek
day = dat.DayOfWeek
MessageBox.Show(day.ToString)         ' zeigt "Thursday"
```

BEISPIEL 4.107: Der wievielte Tag im Jahr ist der 23.8.2012?

VB

```
Dim dat As Date = #8/23/2012#
Dim i As Integer
i = dat.DayOfYear                      ' i erhält den Wert 236
```

BEISPIEL 4.108: Das Datum, was genau 2,5 Tage vor dem 18.6.2010, 18:30 liegt

VB

```
Dim dat As Date = #6/18/2010 6:30:00 PM#
dat = dat.AddDays(-2.5)
MessageBox.Show(dat.ToString)          ' zeigt "16.06.2010 06:30:00"
```

BEISPIEL 4.109: Formatierte Ausgabe der Zeit

VB
```
Dim dat As Date = #6/18/2010 6:30:00 PM#
MessageBox.Show(dat.ToShortTimeString)   ' zeigt "18:30"
```

Wichtige Klasseneigenschaften und -methoden

Die Tabelle zeigt die wichtigsten, dabei ist auch ein alter Bekannter, die *Now*-Methode.

Eigenschaft/Methode	Erläuterung
`Now`	... liefert das aktuelle Datum mit Zeit
`Today`	... liefert aktuelles Datum ohne Zeit
`DaysInMonth(year As Integer, month As Integer) As Integer`	... liefert die Anzahl von Tagen eines Monats
`IsLeapYear(year As Integer) As Boolean`	... stellt fest, ob das Jahr ein Schaltjahr ist
`Parse(s As String[...]) As DateTime`	... dient zum Umwandeln eines Strings in einen *DateTime*-Wert mit optionalen Kulturinformationen

HINWEIS: Als Alternative zur *CDate*-Konvertierungsfunktion lässt sich auch mit der statischen *Parse*-Methode ein *String* in einen *DateTime*-Wert umwandeln.

BEISPIEL 4.110: Umwandlung des Inhalts einer *TextBox* in einen *DateTime*-Wert

VB
```
Dim d As DateTime = DateTime.Parse(TextBox1.Text)
```

Die *Parse*-Methode hat den Vorteil, dass in einer weiteren Überladung zusätzlich Kulturinformationen mit übergeben werden dürfen, welche die Besonderheiten eines bestimmten Landes berücksichtigen.

4.6 Zahlen formatieren

Um Daten in Ihrem Programm anzuzeigen, müssen diese meistens vorher in den *String*-Datentyp verwandelt werden. In den meisten Fällen brauchen Sie für die Datenanzeige die hohe Genauigkeit nicht, mit denen VB 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 Die 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*)

BEISPIEL 4.111: Formatierte Währungsanzeige

VB
```
Private euro As Decimal
TextBox1.Text = euro.ToString("#,##0.00")    ' zeigt z.B. "25.678,50"
```

Für viele übliche Formatierungen können Sie auch standardisierte Formatstrings einsetzen.

BEISPIEL 4.112: Den Formatierungsstring ersetzen wir durch die Standardformatierung "C", die sich auf das in der Systemsteuerung eingestellte Währungs-format bezieht.

VB
```
TextBox1.Text = euro.ToString("C")
```

25.678,50 €

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

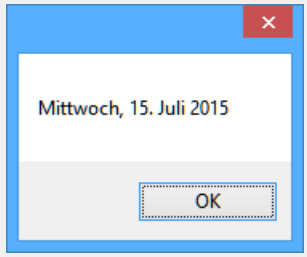
Datumsformatierungen

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.

BEISPIEL 4.113: Das aktuelle Datum wird in einem Meldungsfenster angezeigt.

VB
```
MessageBox.Show(Date.Now.ToString("D"))
```

Ergebnis



Die folgende Tabelle gibt eine Übersicht über die wichtigsten Möglichkeiten.

Formatstring	Beschreibung	Beispiel
d	kurzes Datumsformat	18.06.2015
D	langes Datumsformat	Freitag, 24.August 2015
f	langes Datum mit kurzer Zeit	Freitag, 24.August 2015 08:39
F	langes Datum mit langer Zeit	Freitag, 24.August 2015 08:39:05
g	kurzes Datum mit kurzer Zeit	24.08.2015 08:39
G	kurzes Datum mit langer Zeit	24.08.2015 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	2015-08-24 08:39:05Z
`y` oder `Y`	Jahr Monat	August 2015
dd.MM.yy	nutzerdefiniert	24.08.12
yyyy-MM-dd	dto.	2015-08-24
MM\/dd\/yy	dto.	08/24/12
hh mm:ss	dto.	08:39:05
h:mm	dto.	8:39

HINWEIS: Informationen zu Datums-/Zeitfunktionen finden Sie im Abschnitt 4.4.

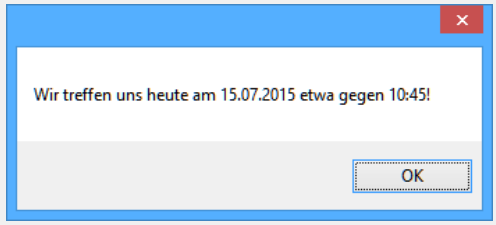
4.6.2 Die Format-Methode

Hier handelt es sich um eine statische Methode der *String*-Klasse, mit welcher man ziemlich komplexe Formatierungen erzielen kann.

BEISPIEL 4.114: Zwei Datumsformatierungen sind in einem Text eingeschlossen.

VB
```
Dim st As String = "Wir treffen uns heute am {0:d} etwa gegen {1:h:mm}!"
MessageBox.Show(String.Format(st, Date.Now, #10:45 AM#))
```

Ergebnis



Zu obigem Beispiel gibt es sicherlich allerhand Klärungsbedarf:

Der String *st* enthält in den geschweiften Klammern zwei Formatspezifikationen, deren allgemeine Syntax wie folgt ist:

SYNTAX: `{Index[, Minimalbreite][:Formatstring]}`

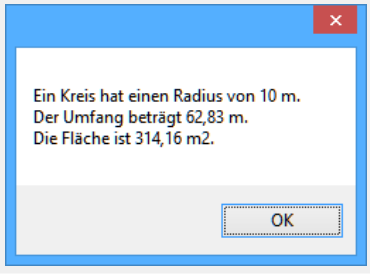
- Der **Index** bestimmt das zu formatierende Argument. In unserem Beispiel gibt es zwei Argumente, *d* und *#10:45 AM#*, das erste hat wie immer den Index 0, das zweite den Index 1.
- Mit **Minimalbreite** können Sie optional angeben, wie breit der formatierte Teilstring mindestens sein soll. Für positive 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.

Ein weiteres Beispiel aus dem mathematischen Bereich soll unsere Betrachtungen zur *Format*-Methode abrunden (weitere umfangreiche Informationen finden Sie in der Hilfe-Dokumentation).

BEISPIEL 4.115: Hier werden drei Argumente berechnet und formatiert.

VB
```
Dim r As Integer = 10
Dim s As String = "Ein Kreis hat einen Radius von {0} m." & vbCrLf &
                  "Der Umfang beträgt {1:0.00} m." & vbCrLf &
                  "Die Fläche ist {2:0.00} m2."
MessageBox.Show(String.Format(s, r, 2 * Math.PI * r, Math.PI * r * r))
```

Ergebnis



4.6.3 Stringinterpolation

Seit VB 2015 führt noch einen deutlich einfacherer Weg zum Formatstring. Dabei ist dem String lediglich ein Dollarzeichen voranzustellen. Die variablen Werte können dann direkt eingetragen werden.

BEISPIEL 4.116: Der (deutsche) Wochentag des aktuellen Datums wird angezeigt[1]

```
VB  Label1.Text = $"{DateTime.Now:dddd}"      ' liefert z.B. "Dienstag"
```

Die beiden Beispiele des Vorgängerabschnitts lassen sich drastisch vereinfachen:

BEISPIEL 4.117: Komplexe Datumsformatierung (vereinfacht)

```
VB  MessageBox.Show($"Wir treffen uns heute am {DateTime.Now.Date:d} etwa gegen 10:45!")
```

BEISPIEL 4.118: Kreisberechnung (vereinfacht)

```
VB  Dim s As String = $"Ein Kreis hat einen Radius von {(r):0.00} m." & vbCrLf &
                      $"Der Umfang beträgt {(2 * Pi):0.00} m." & vbCrLf &
                      $"Die Fläche ist {(Pi * r * r):0.00} m2"
```

4.7 Praxisbeispiele

4.7.1 Zeichenketten verarbeiten

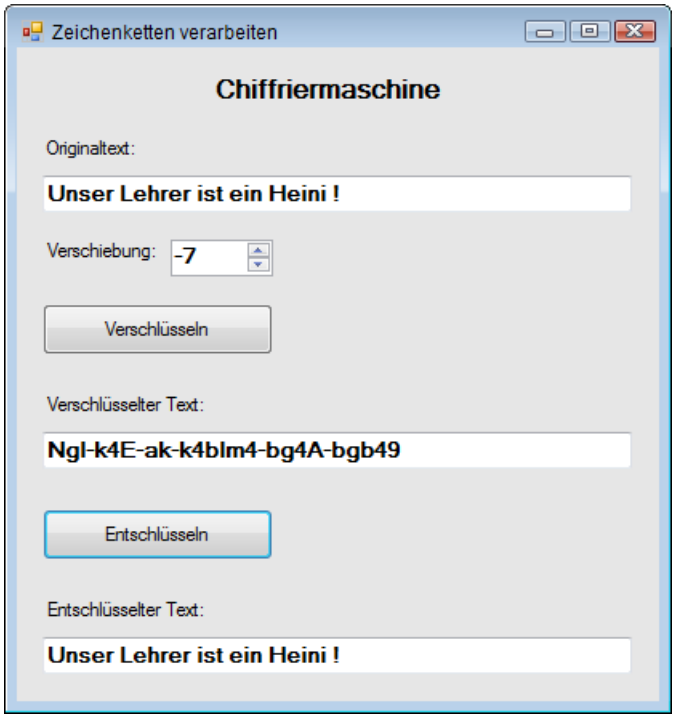
Die zahlreichen Eigenschaften und Methoden von Stringvariablen sollen an folgendem, nicht ganz ernst zu nehmenden, Beispiel demonstriert werden:

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*):

[1] Die *DayOfWeek*-Methode liefert nur den englischen Wochentag.



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 Class Form1
```

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 s0 As String =
                "abcdefghijklmnopqrstuvwxyzäöüABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ1234567890 .,-?!"
     Private nmax As Integer = s0.Length
```

Beim Laden des Formulars werden der obere und untere Grenzwert der *NumericUpDown*-Komponente auf die positive bzw. negative Länge des Zeichensatzes gesetzt:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        NumericUpDown1.Maximum = nmax
```

```
        NumericUpDown1.Minimum = -nmax
    End Sub
```

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 Function codieren(s As String, n As Integer) As String
        Dim s1 As String = String.Empty         ' zurückgegebener 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":

```
        Dim pos As Integer             ' Originalposition des Zeichens im Zeichensatz
        Dim posN As Integer            ' Position des verschobenen Zeichens im Zeichensatz
        Dim z As Char                  ' einzelnes Zeichen
        For i As Integer = 0 To s.Length - 1     ' für alle Zeichen des übergebenen Strings
            z = CChar(s.Substring(i))            ' i-tes Zeichen herausschneiden
            pos = s0.IndexOf(z)      ' Position im Zeichensatz suchen
            If pos = -1 Then         ' Zeichen nicht gefunden
                MessageBox.Show(z & " ist ein unzulässiges Zeichen!", "Warnung")
                Return String.Empty
            End If
            posN = pos + n             ' Verschiebeposition berechnen
            If posN > nmax Then posN = posN - nmax       ' bei Überlauf wieder vorn
            If posN < 0 Then posN = posN + nmax          ' ... bzw. hinten weitermachen
            z = CChar(s0.Substring(posN))                ' korrespondierendes Zeichen ermitteln
            s1 = s1 & z                ' Rückgabestring zusammensetzen
        Next i
        Return s1
    End Function
```

Der Funktionsaufruf beim Verschlüsseln:

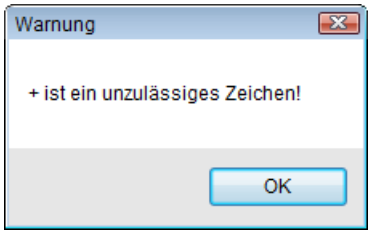
```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        TextBox2.Text = codieren(TextBox1.Text, CInt(NumericUpDown1.Value))
    End Sub
```

Der Aufruf beim Entschlüsseln:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        TextBox3.Text = codieren(TextBox2.Text, CInt(-NumericUpDown1.Value))
    End Sub
End Class
```

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 der 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 etwas entschärfen lässt.

4.7.2 Zeichenketten mittels StringBuilder addieren

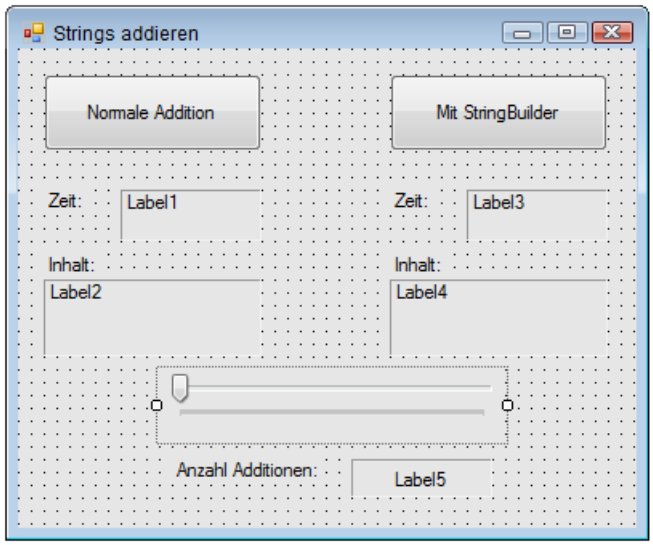
Unter .NET sind auch Strings Referenztypen, d h., eine Stringvariable speichert nicht den Wert, sondern lediglich einen Verweis (Referenz, Adresse) auf die Speicherstelle. Allerdings muss ein wichtiger Unterschied beachtet werden:

HINWEIS: Strings werden im Speicher als **unveränderliche** Zeichenketten abgelegt. Mit jeder Änderung einer String-Variablen wird ein **neuer String** erzeugt!

In diesem Beispiel 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 Zeitmessung.

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie die abgebildete Benutzerschnittstelle. Weisen Sie der *TrackBar* die Werte *Maximum = 50000, Minimum = 1000, SmallChange = 1000, LargeChange = 10000* zu.



Quellcode Klasse CTest

Über das Menü *Projekt|Klasse hinzufügen...* fügen Sie zum Projekt eine neue Datei mit dem Namen *Test.vb* hinzu.

```
Imports System.Text
```

Die Klasse *CTest* stellt zwei statische 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 Structure TestResult
    Dim inhalt As String
    Dim zeit As Integer
End Structure

Public 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 Shared Function addNormal(nr As Integer) As TestResult
        Dim t1 As DateTime = DateTime.Now
        Dim r As TestResult
```

String addieren:

```
        Dim str As String = String.Empty
```

```
        For i As Integer = 0 To nr - 1
            str &= i.ToString & " "
        Next i
        r.inhalt = str
```

Zeitmessung:

```
        Dim t2 As DateTime = DateTime.Now
        Dim ts As New TimeSpan(t2.Ticks - t1.Ticks)
        r.zeit = Convert.ToInt32(ts.TotalMilliseconds)
        Return r
    End Function
```

Analog ist die Methode *addWithStringBuilder* aufgebaut, welche zur Addition die *Append*-Methode der *StringBuilder*-Klasse benutzt:

```
    Public Shared Function addWithStringBuilder(nr As Integer) As TestResult
        Dim t1 As DateTime = DateTime.Now
        Dim r As TestResult
```

Mit *StringBuilder* addieren:

```
        Dim sb As New StringBuilder()
        For i As Integer = 0 To nr - 1
            sb.Append(i.ToString & " ")
        Next i
        r.inhalt = sb.ToString
```

Zeitmessung:

```
        Dim t2 As DateTime = DateTime.Now
        Dim ts As New TimeSpan(t2.Ticks - t1.Ticks)
        r.zeit = Convert.ToInt32(ts.TotalMilliseconds)
        Return r
    End Function
End Class
```

Quellcode Form1

```
Public Class Form1
```

Die normale Stringaddition:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Label1.Text = String.Empty
        Label2.Text = String.Empty
        Me.Refresh()
        Me.Cursor = Cursors.WaitCursor
        Dim tr As TestResult = CTest.addNormal(TrackBar1.Value)
        Label1.Text = tr.zeit.ToString & " ms"
        Dim s As String = 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)
        Me.Cursor = Cursors.Default
    End Sub
```

Die Addition mittels *StringBuilder*:

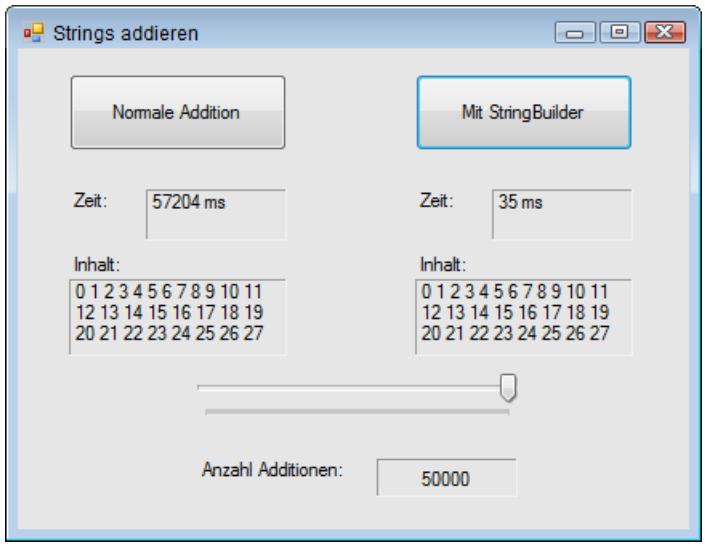
```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Label3.Text = String.Empty :  Label4.Text = String.Empty
        Me.Refresh()
        Me.Cursor = Cursors.WaitCursor
        Dim tr As TestResult = CTest.addWithStringBuilder(TrackBar1.Value)
        Label3.Text = tr.zeit.ToString & " ms"
        Dim s As String = tr.inhalt
        Label4.Text = s.Substring(0, 100)
        Me.Cursor = Cursors.Default
    End Sub
```

Die Anzeige der Anzahl der Schleifendurchläufe wird aktualisiert:

```
    Private Sub TrackBar1_ValueChanged(sender As Object, e As EventArgs) _
                        Handles TrackBar1.ValueChanged
        Label5.Text = TrackBar1.Value.ToString
    End Sub
End Class
```

Test

Kompilieren Sie das Programm und stellen Sie zuerst die gewünschte Anzahl von Additionen ein. Das Ergebnis ist eindrucksvoll.



Wie das abgebildete Beispiel zeigt, bringt hier bei 50000 Durchläufen die Verwendung eines *StringBuilders* einen Performancegewinn fast um 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 *StringBuilder*s 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

Siehe Laufzeitansicht.

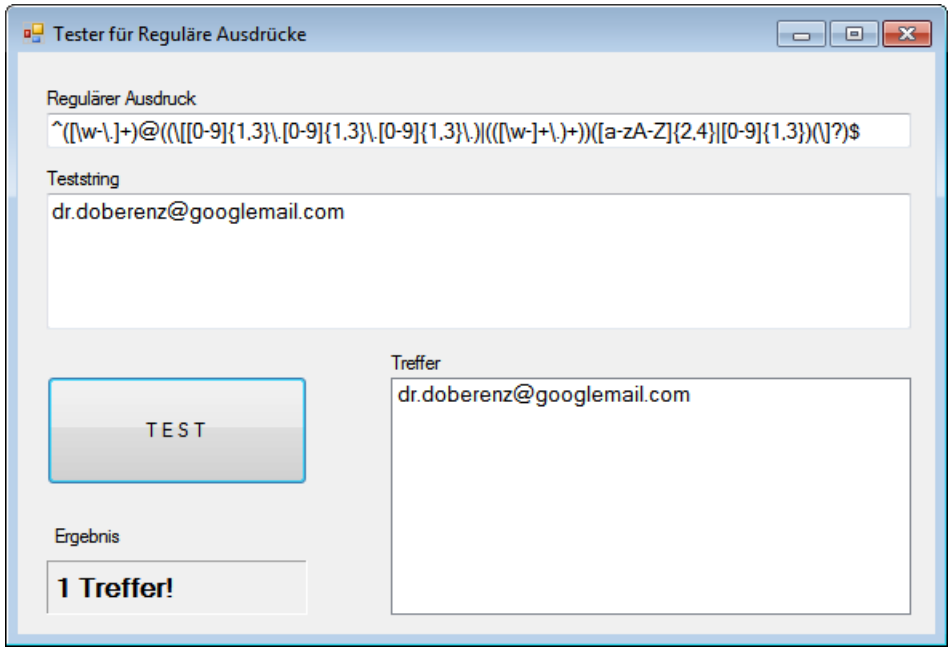
Quellcode

```
...
Imports System.Text.RegularExpressions

Public Class Form1
    Private rex As String = Nothing
    Private text As String = Nothing

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        rex = TextBox1.Text
        txt = TextBox2.Text
        ListBox1.Items.Clear()
        Dim rxcoll As MatchCollection = Regex.Matches(txt, rex)
        Label1.Text = rxcoll.Count.ToString & " Treffer!"
        For Each m As Match In rxcoll
```

```
            ListBox1.Items.Add(m.Value)
        Next
    End Sub
End Class
```

Test

In die *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 VB-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!

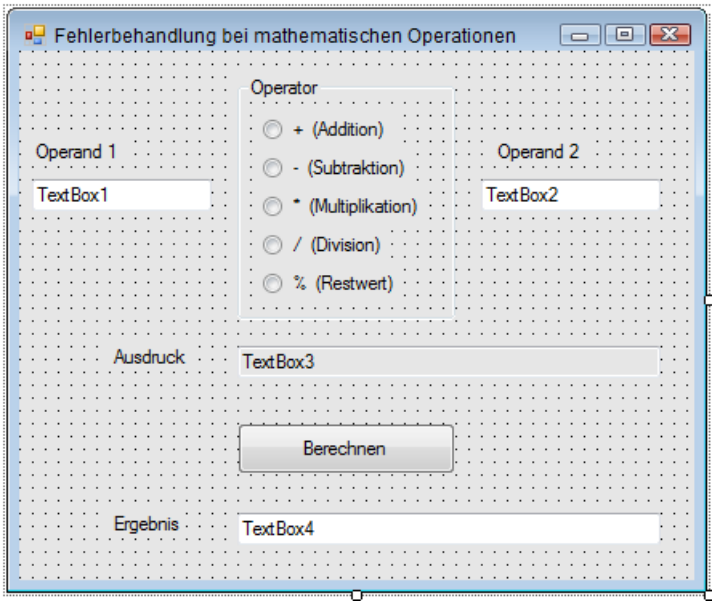
Eine weitere interessante Anwendung finden Sie im Praxisteil von Kapitel 7!

4.7.4 Fehler bei mathematischen Operationen behandeln

Dieses Rezept soll nur nebenbei die (doch relativ triviale) Anwendung der mathematischen Grundoperationen demonstrieren. Vor allem soll auf die Möglichkeiten der Fehlerbehandlung mittel *Try-Catch*-Blöcken und *Throw Exception* hingewiesen werden.

Oberfläche

Die Abbildung zeigt einen sehr einfachen "Taschenrechner", der nur für ganze Zahlen (Integer) ausgelegt ist.



In einer *GroupBox* sind fünf *RadioButton*s angeordnet. Das Textfeld für die Ergebnisanzeige (*TextBox4*) ist deshalb so breit, weil dort auch die eventuellen Fehlermeldungen erscheinen sollen.

Quellcode

```
Public Class Form1
```

Für jede der fünf arithmetischen Grundoperationen wird eine eigene Funktion definiert, welche die Anzeige des Ausdrucks übernimmt und die Berechnung durchführt.

Alle fünf Operationen nehmen nur *Integer*-Werte als Operanden entgegen und liefern als Ergebnis einen Integer-Wert zurück!

Die Addition:

```
    Private Function addiere(op1 As Integer, op2 As Integer) As Integer
        TextBox3.Text = op1.ToString() & " + " & op2.ToString()
```

```
        Return op1 + op2
    End Function
```

Die Subtraktion:

```
    Private Function subtrahiere(op1 As Integer, op2 As Integer) As Integer
        TextBox3.Text = op1.ToString() & " - " & op2.ToString()
        Return op1 - op2
    End Function
```

Das Multiplizieren:

```
    Private Function multipliziere(op1 As Integer, op2 As Integer) As Integer
        TextBox3.Text = op1.ToString() & " * " & op2.ToString()
        Return op1 * op2
    End Function
```

Die Integer-Division:

```
    Private Function dividiere(op1 As Integer, op2 As Integer) As Integer
        TextBox3.Text = op1.ToString() & " / " & op2.ToString()
        Return (op1 \ op2)
    End Function
```

Die Restwert-Division:

```
    Private Function modulo(op1 As Integer, op2 As Integer) As Integer
        TextBox3.Text = op1.ToString() & " % " & op2.ToString()
        Return op1 Mod op2
    End Function
```

In Abhängigkeit vom angeklickten *RadioButton* führt die folgende Funktion die Operation aus bzw. liefert eine Fehlermeldung:

```
    Private Function berechne(op1 As Integer, op2 As Integer) As Integer
        Dim res As Integer = 0
        If RadioButton1.Checked Then
            res = addiere(op1, op2)
        ElseIf (RadioButton2.Checked) Then
            res = subtrahiere(op1, op2)
        ElseIf (RadioButton3.Checked) Then
            res = multipliziere(op1, op2)
        ElseIf (RadioButton4.Checked) Then
            res = dividiere(op1, op2)
        ElseIf (RadioButton5.Checked) Then
            res = modulo(op1, op2)
        Else
            Throw New InvalidOperationException("Kein Operator ausgewählt!")
        End If
        Return res
    End Function
```

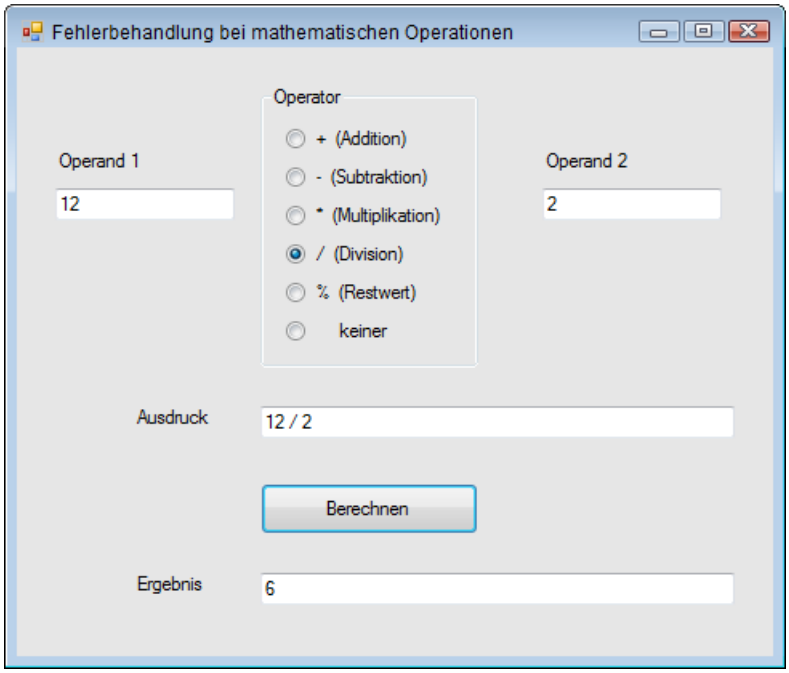
Die Schaltfläche "Berechnen":

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Try
            Dim op1 As Integer = System.Int32.Parse(TextBox1.Text)
            Dim op2 As Integer = System.Int32.Parse(TextBox2.Text)
            Dim res As Integer = berechne(op1, op2)
            TextBox4.Text = res.ToString()
        Catch ex As System.Exception
            TextBox4.Text = ex.Message
        End Try
    End Sub
End Class
```

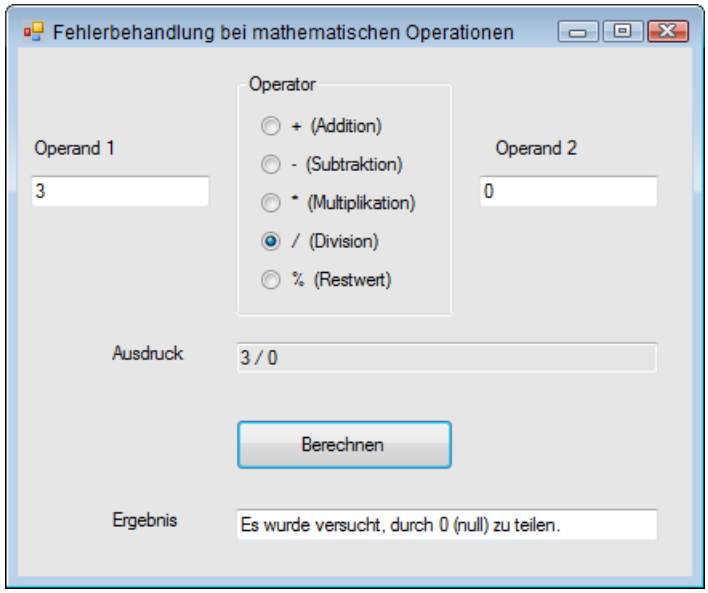
HINWEIS: Achten Sie darauf, dass im obigen Code an zwei unterschiedlichen Stellen eine Fehlerbehandlung erfolgt (Fettdruck)!

Test

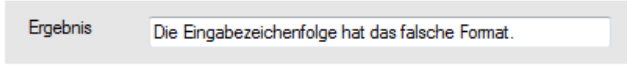
Ihrer Experimentierfreude sind nun keine Grenzen gesetzt, dank der doppelten Fehlerbehandlung werden Sie großzügig mit Informationen versorgt und es wird Ihnen nicht gelingen, das Programm zum Absturz zu bewegen.



Auch bei der gefürchteten Division durch Null erfolgt anstelle des Ergebnisses eine vernünftige Fehlermeldung:



Weitere Fehlermeldungen:



Bemerkung

Sie können leicht nachprüfen, dass das Ergebnis der Modulo-Division gleich dem Restwert ist, wie er bei Integer-Division übrig bleibt, z.B.:

```
10 \ 3 = 3
```

Der Rest ist 1:

```
10 Mod 3 = 1
```

4.7.5 Methodenaufrufe mit Array-Parametern

Für eine Liste von Zahlen wollen wir den *arithmetischen Mittelwert*, den *geometrischen Mittelwert* und den *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:

- *MeanAritmetic* (für den arithmetischen Mittelwert)
- *MeanGeometric* (für den geometrischen Mittelwert)
- *MeanMedian* (für den Wert in der Mitte der sortierten Folge)

```
Public Class Form1
```

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 enumAvType As Byte
        MeanA = 0               ' arithmetisches Mittel
        MeanG                   ' geometrisches Mittel
        Median                  ' Wert in der Mitte
    End Enum
```

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 Function MeanArithmetic(decArray() As Decimal) As Decimal
        Dim anzahl As Integer = decArray.GetLength(0)
        Dim summe As Decimal = 0
        Dim i As Integer = 0
        If anzahl = 0 Then Return 0
        While i <= decArray.GetUpperBound(0)
            summe += decArray(i)
            i += 1
        End While
        Return (summe / anzahl)
    End Function
```

MeanGeometric-Funktion: Die n-te Wurzel aus dem Produkt aller Zahlen (geometrischer Mittelwert).

```
    Private Function MeanGeometric(arr() As Decimal) As Decimal
        Dim total As Double = 1
        Dim n As Integer = arr.GetLength(0)
        For i As Integer = 0 To n - 1
            total *= Convert.ToDouble(arr(i))
```

```
        Next
        Return Convert.ToDecimal(Math.Pow(total, 1 / n))    ' n-te Wurzel aus total
    End Function
```

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 Function MeanMedian(arr() As Decimal) As Decimal
        Dim anzahl As Integer = arr.GetLength(0)
        If anzahl = 0 Then Return 0
        Array.Sort(arr)                ' Zahlen  sortieren!
        If anzahl Mod 2 = 0 Then       ' geradzahlig
            Return (arr(CInt(anzahl / 2) - 1) + arr(CInt(anzahl / 2))) / 2
        Else                          ' ungeradzahlig
            Return arr(anzahl \ 2)   ' Integer-Division!
        End If
    End Function
```

Die Hauptmethode *Average* entscheidet auf Grundlage der übergebenen Enumerationskonstanten, zu welcher der drei speziellen Funktionen verzweigt werden soll:

```
    Public Function Average(arr() As Decimal,
       Optional AvType As enumAvType = enumAvType.MeanA) As Decimal
        Select Case AvType
            Case enumAvType.MeanA
                Return MeanArithmetic(arr)
            Case enumAvType.MeanG
                Return MeanGeometric(arr)
            Case enumAvType.Median
                Return MeanMedian(arr)
            Case Else
                Throw New Exception("Falscher Mittelwert-Typ (" & AvType.ToString & ").")
        End Select
    End Function
```

Nun endlich kommen wir zur Anwendung (Klick auf den "Start"-Button):

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim n As Integer = ListBox1.Items.Count
        Dim arr(n - 1) As Decimal
        For i As Integer = 0 To n - 1
            arr(i) = CDec(ListBox1.Items(i))
        Next i
```

Dreimaliger Aufruf der *Average*-Funktion mit unterschiedlichen Konstanten:

```
        Label1.Text = Average(arr, enumAvType.MeanA).ToString("#.000")    ' arithm. MW
        Label2.Text = Average(arr, enumAvType.MeanG).ToString("#.000")    ' geometr. MWEnter
        Label3.Text = Average(arr, enumAvType.Median).ToString            ' Wert in Mitte
    End Sub
```

Um eine Zahl zur *ListBox* hinzuzufügen, geben Sie diese in die *TextBox* ein und schließen mit der Enter-Taste ab:

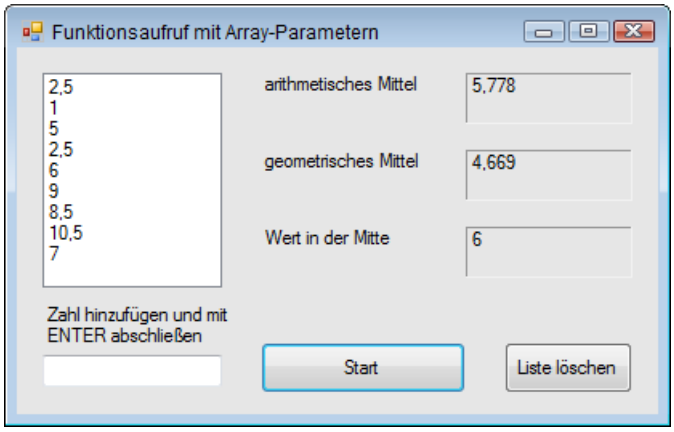
```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter Then
            ListBox1.Items.Add(TextBox1.Text)
            TextBox1.Clear()
        End If
    End Sub
```

Die *ListBox* löschen:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        ListBox1.Items.Clear()
        Label1.Text = String.Empty : Label2.Text = String.Empty : Label3.Text = String.Empty
    End Sub
End Class
```

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

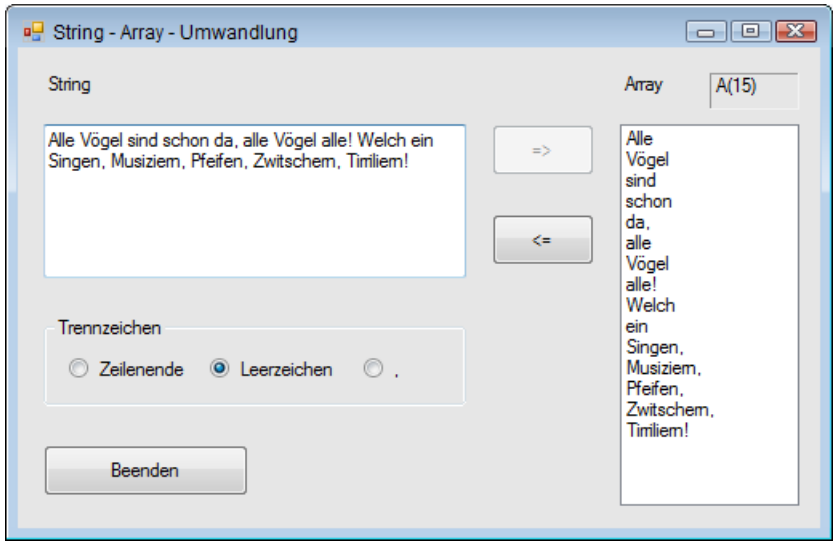
- Verwenden Sie zur Zahleneingabe das Dezimalkomma und nicht den Dezimalpunkt.
- Jede Eingabe ist mittels *Enter*-Taste abzuschließen.

4.7.6 String in Array kopieren und umgekehrt

Es gibt die verschiedensten Gründe, einen String zu zerpflücken und seine Bestandteile in einem Array abzuspeichern (Formelparser, Compiler, Suchfunktionen, ...). Dieses Rezept zeigt die Vorgehensweise und dürfte gleichzeitig ein gutes Beispiel für das Zusammenwirken von Instanzen- und Klassenmethoden sein (*Split* und *Join*).

Oberfläche

Um die Wirkungsweise anschaulich zu demonstrieren, zeigen wir den Stringinhalt links in einer *TextBox* an und den Arrayinhalt rechts in einer *ListBox* (siehe Laufzeitsansicht). Die drei *RadioButton*s dienen uns zum Einstellen des gewünschten Trennzeichens (Zeilenumbruch, Leerzeichen oder Komma). Die Abbildung zeigt die Laufzeitansicht.



Quellcode

```
Public Class Form1
```

Das Trennzeichen bestimmt die Aufteilung des Strings:

```
    Private tz As Char
```

Ein dynamisches Array soll als Behälter für die Bestandteile des Strings dienen:

```
    Private A() As String
```

Damit Sie sich das mühselige Eintippen eines Beispieltextes ersparen, wird die Textbox bereits beim Laden des Formulars gefüllt:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        TextBox1.Text = "Alle Vögel sind schon da, alle Vögel alle! "
        TextBox1.Text &= "Welch ein Singen, Musiziern, "
        TextBox1.Text &= "Pfeifen, Zwitschern, Tirriliern!"
    End Sub
```

Zum Kopieren des Strings in das Array wird die *Split*-Methode des Strings ausgeführt, der man als Parameter das Trennzeichen übergibt. Rückgabewert ist das gefüllte Array.

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim s As String = TextBox1.Text
```

Die Auswahl des Trennzeichens (nur erstes Zeichen nehmen):

```
        If radioButton1.Checked Then tz = CChar(Environment.NewLine)
        If radioButton2.Checked Then tz = CChar(" ")
        If radioButton3.Checked Then tz = CChar(",")
        TextBox1.Text = String.Empty
```

Hier erfolgt die Trennung:

```
        A = s.Split(tz)
```

Die Anzeige des Arrayinhalts dient lediglich unserer Information:

```
        ListBox1.Items.Clear()
        For i As Integer = 0 To A.Length - 1
            ListBox1.Items.Add(A(i))
        Next
```

Ebenfalls nur der Befriedigung unserer Neugier dient die Anzeige der Arraygröße:

```
        Label1.Text = "A(" & A.Length.ToString & ")"
    End Sub
```

Die Rückverwandlung des Arrays in einen String ist mit der (statischen) *Join*-Methode der *String*-Klasse eine einfache Angelegenheit. Übergabeparameter sind das Trennzeichen und das gefüllte Array. Rückgabewert ist der zusammengesetzte String:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        ListBox1.Items.Clear()
        Dim s As String = String.Join(tz, A)
        TextBox1.Text = s
    End Sub
End Class
```

Test

Kopieren Sie die Strings unter Verwendung verschiedener Trennzeichen hin und zurück (siehe obige Laufzeitansicht).

4.7.7 Ein Byte-Array in einen String konvertieren

Haben Sie beispielsweise ein Byte-Array mit acht Feldern und möchten die ersten fünf Felder in einen ersten String und die letzten in einen zweiten String kopieren, so sind Sie als Einsteiger ratlos. Als "richtiger" Programmierer wissen Sie sich natürlich zu helfen. Getreu der Devise "Hauptsache es funktioniert" werden Sie mit den Ihnen bekannten elementaren String-Funktionen natürlich zu einer Lösung kommen.

Erst wenn Sie sich ein wenig in der gigantischen Klassenbibliothek des .NET-Frameworks auskennen, wird Ihnen vielleicht eine elegantere Lösung einfallen und Sie wissen, dass die *Encoding*-Klasse Methoden bereitstellt, mit deren Hilfe Arrays und Zeichenfolgen von Unicode-Zeichen aus direkt in Byte-Arrays konvertiert werden können, die für eine bestimmte Zielcodepage codiert sind.

Das vorliegende Praxisbeispiel bietet durch Gegenüberstellung beider Varianten gute Vergleichsmöglichkeiten.

Oberfläche

Zwei Schaltflächen und zwei *Label* genügen für einen Test (siehe Laufzeitabbildung).

Quelltext

```
Public Class Form1
```

Die Quelle ist ein gefülltes Byte-Array, welches den Zeichen *A, B, C, D, E, F, G, H* entspricht:

```
    Private ba() As Byte = {65, 66, 67, 68, 69, 70, 71, 72}
```

Das Ziel sind zwei Strings, in welche die ersten fünf bzw. die letzten drei Elemente des Arrays kopiert werden sollen:

```
    Private s1, s2 As String
```

Beginnen wir mit der umständlichen Lösung, bei der wir jedes einzelne Byte in einer *For-Next*-Schleife aus dem Array herauskopieren und mittels explizitem Typecasting in einen *Char*-Datentyp verwandeln und zum Ergebnisstring hinzuaddieren:

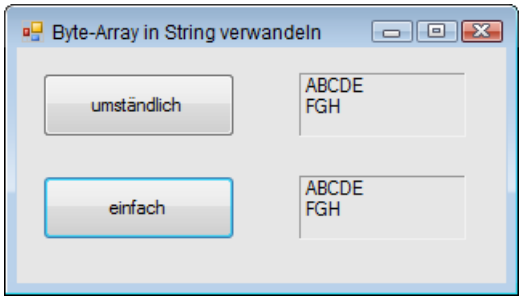
```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim i As Integer
        For i = 0 To 4
            s1 &= Convert.ToChar(ba(i))
        Next i
        For i = 5 To 7
            s2 &= Convert.ToChar(ba(i))
            ' s2 &= CChar(ba(i))       ' geht nicht!
            ' s2 &= CType(ba(i), Char) ' geht nicht!
        Next i
        Label1.Text = s1 & Environment.NewLine & s2
    End Sub
```

Jetzt die einfache Lösung, bei der jeweils eine Anweisung genügt, um mittels *GetString*-Methode das Byte-Array zu decodieren:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        ' decodiert Byte-Array:
        s1 = System.Text.Encoding.Default.GetString(ba, 0, 5)   ' 5 Zeichen ab Position 0
        s2 = System.Text.Encoding.Default.GetString(ba, 5, 3)   ' 3 Zeichen ab Position 5
        Label2.Text = s1 & Environment.NewLine & s2
    End Sub
End Class
```

Test

Die aus dem *Byte*-Array erzeugten Strings werden rechts angezeigt. Wie man sieht, kommen beide Varianten zum gleichen Ergebnis (siehe folgende Abbildung).

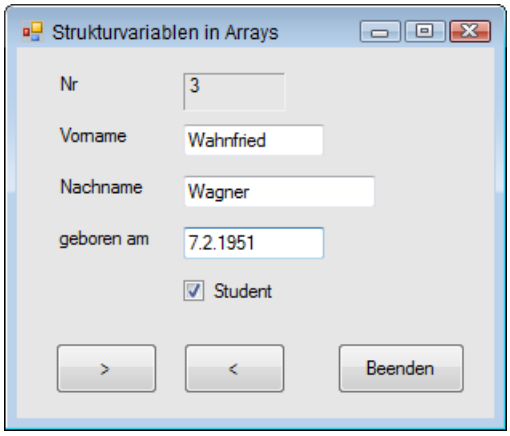


4.7.8 Strukturvariablen in Arrays einsetzen

Dieses Praxisbeispiel zeigt dem Einsteiger anschaulich, wie man ein Array von Strukturvariablen erzeugen und darauf zugreifen kann. Ziel ist eine kleine Personalverwaltung, in welcher wir Daten von maximal zehn Personen ablegen wollen.

HINWEIS: Strukturvariablen kann man durchaus als Vorläufer der Objekte bezeichnen. Wer ihre Verwendung beherrscht, dem wird auch der Übergang zur OOP (Objektorientierte Programmierung) nicht schwer fallen.

Oberfläche



Die obige Abbildung zeigt die Laufzeitansicht der aus drei *TextBox*en, einer *CheckBox* und mehreren *Label*s und *Button*s bestehenden Bedienoberfläche.

Quellcode

```
Public Class Form1
```

Die Struktur definieren:

```
    Private Structure Person
        Dim Vorname, Nachname As String
        Dim Geburt As Date
        Dim Student As Boolean
    End Structure
```

Einige globale Variablen:

```
     Const pmax As Integer = 10                ' maximale Anzahl Personen
     Private pListe(pmax) As Person            ' statisches Array aus Strukturvariablen
     Private pos As Integer = 1                ' Positionszähler
```

Die Methode zum Anzeigen eines Datensatzes:

```
    Private Sub anzeigen()
        Label1.Text = pos.ToString
        With pListe(pos)
            TextBox1.Text = .Vorname
            TextBox2.Text = .Nachname
            TextBox3.Text = CStr(.Geburt)
            CheckBox1.Checked = .Student
        End With
    End Sub
```

Die Methode zum Speichern eines Datensatzes:

```
    Private Sub speichern()
        With pListe(pos)
            .Vorname = TextBox1.Text
            .Nachname = TextBox2.Text
            .Geburt = CDate(TextBox3.Text)
            .Student = CheckBox1.Checked
        End With
    End Sub
```

Es folgen vier Ereignisbehandlungsroutinen (Eventhandler).

Beim Laden des Formulars wird die Anzeige mit Anfangswerten initialisiert:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Label1.Text = "1"
        TextBox1.Text = ""
        TextBox2.Text = ""
        TextBox3.Text = "00:00:00"
    End Sub
```

Vorwärts blättern mit der ">"-Schaltfläche:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        If pos < pmax Then
            Call speichern()
            pos += 1
            Call anzeigen()
        End If
    End Sub
```

Rückwärts blättern mit der "<"-Schaltfläche:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        If pos > 1 Then
            Call speichern()
            pos -= 1
            Call anzeigen()
        End If
    End Sub
    ...
End Class
```

Test

Nach dem Programmstart (F5) können Sie das leere *Personal*-Array mit Daten füllen (siehe obige Laufzeitansicht).

HINWEIS: Die Übernahme in den Speicher erfolgt hier erst **nach** dem Weiterblättern!

Kapitel 5

Weitere Sprachfeatures

Da Sie nun mit den wichtigsten Sprachelementen und OOP-Konzepten vertraut sind, können wir jetzt die sprachlichen Grundlagen von Visual Basic 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.

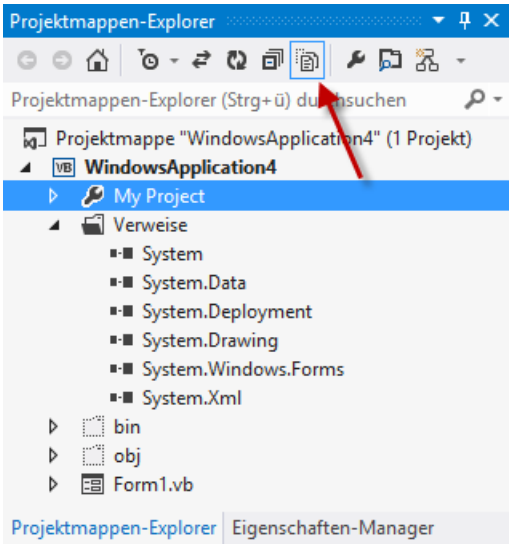
5.1 Namespaces (Namensräume)

Zu Beginn dieses Kapitels wollen wir uns einem Thema widmen, welches unbedingt mit zu den Grundkonzepten von .NET gehört, die Namensräume. Diese 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.

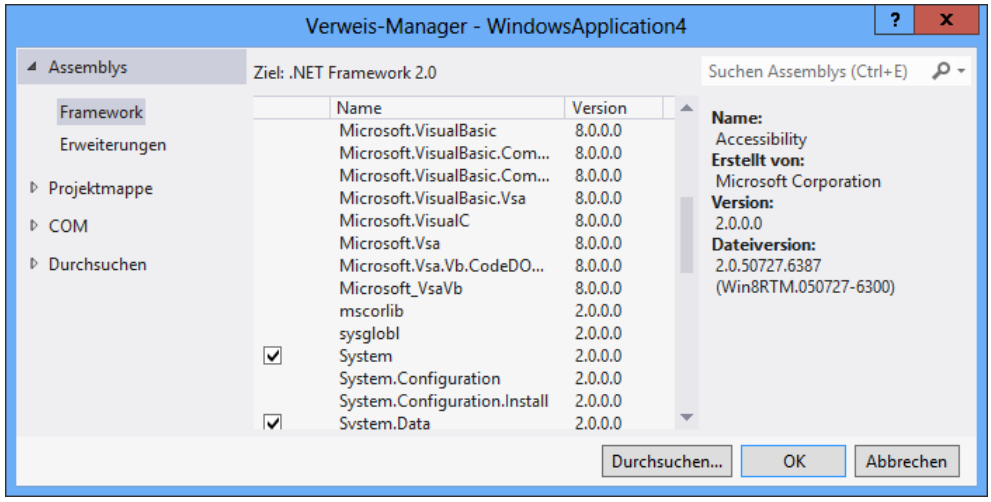
5.1.1 Ein kleiner Überblick

Namensräume bilden 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 (Klicken Sie vorher auf die kleine Schaltfläche *Alle Dateien anzeigen* unterhalb der Titelleiste des Projektmappen-Explorers):



Wenn Sie für Ihre Programmentwicklung noch weitere Klassen benötigen, so öffnen Sie das Kontextmenü *Verweise* im Projektmappen-Explorer und wählen Sie *Verweis hinzufügen...*



In der Liste der Verweise erkennen Sie auch den Pfad zu der Datei, die die entsprechende Klassenbibliothek enthält (**.dll*). Meist sind inhaltlich verwandte Klassen in einem Namespace zusammengefasst. Es ist aber durchaus üblich, dass eine Datei mehrere Namensräume enthält. Auch der umgekehrte Fall, dass sich ein Namespace über mehrere Dateien erstreckt, ist möglich.

5.1.2 Die Imports-Anweisung

Um einem häufigen Missverständnis vorzubeugen, soll gleich zu Beginn eine Klarstellung erfolgen:

HINWEIS: Die *Imports*-Anweisung "importiert" keine Klassen bzw. Namespaces, sie verkürzt nur die Schreibweise beim Zugriff auf bestimmte Klassen bzw. Namespaces, die bereits über *Verweise* (siehe oben) eingebunden sind.

Klassen, die sich im selben 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 (so genannte *voll qualifizierte Bezeichner*). Um derartige "Bandwürmer" zu vermeiden, steht am Anfang eines Projekts bzw. vor einer Klassendefinition in der Regel eine mehr oder weniger lange Liste von *Imports*-Anweisungen, was im nachfolgenden Code zu einer verkürzten Typnotation führt, welche meist nur noch den Typbezeichner erforderlich macht.

BEISPIEL 5.1: Die voll qualifizierte Typdeklaration einer generischen Liste

VB
```
Dim myList As System.Collections.Generic.List(Of String)
```
... kann wie folgt verkürzt werden:
```
Imports System.Collections.Generic
...
Dim myList As List(Of String)
```

Das Haar in der Suppe: Wenn in verschiedenen Namensräumen Klassen mit gleichen Typbezeichnern existieren, so führt die Verwendung von *Imports* 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 die Verwendung eines Alias.

5.1.3 Namespace-Alias

Ausgangspunkt ist die vollständige Syntax der *Imports*-Anweisung:

SYNTAX: `Imports [Alias =] Namespace`

BEISPIEL 5.2: Ein Alias für den Namespace *System.Collections.Generic* vereinfacht die Deklaration einer generischen Auflistung

VB
```
Imports GColl = System.Collections.Generic
...
Dim myList As GColl.List(Of String)
```

Man kann das sogar noch weiter treiben und selbst komplette Klassenbezeichner durch einen Alias ersetzen.

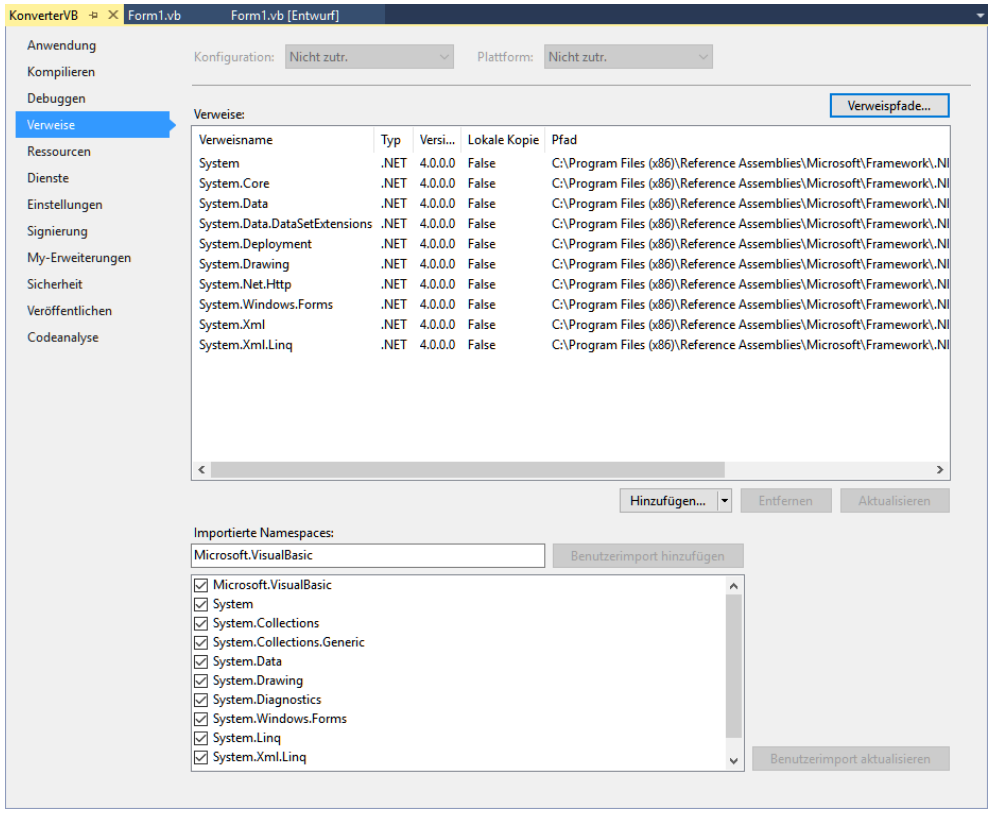
BEISPIEL 5.3: Eine Modifikation des Vorgängerbeispiels

```
VB
Imports GCollStrList As System.Collections.Generic.List(Of String)
...
Dim myList As GCollStrList
```

5.1.4 Namespaces in Projekteigenschaften

Öffnen Sie das Eigenschaftenfenster des Projekts (Menü *Projekt/<Anwendungsname>-Eigenschaften...* bzw. Kontextmenü *Eigenschaften* oder Doppelklick auf den Knoten *MyProject* im Projektmappen-Explorer), so können Sie sich auf der Seite "Verweise" einen Überblick über die eingebundenen Verweise (oben) und die anwendungsweit gültigen Namespaces (unten) verschaffen. Die Liste der Verweise entspricht den im Projektmappen-Explorer aufgeführten Verweisen.

Durch Setzen von Häkchen bei "Importierte Namespaces" können Sie Namensräume einbinden, die dann für die gesamte Anwendung gültig sind.



5.1.5 Namespace Alias Qualifizierer

Die Fähigkeit des Zugriffs auf ein Mitglied im globalen Namespace ist nützlich wenn ein Mitglied möglicherweise von einem anderen mit gleichem Namen verdeckt wird. Das führt dazu, dass zunächst im globalen Namespace "nachgeschaut" wird, anstatt im direkt zugeordneten Namespace.

Das Schlüsselwort *Global* ermöglicht den Zugriff auf den Stamm-Namespace (oder leeren Namespace) an erster Stelle in der Namespace-Hierarchie.

BEISPIEL 5.4: *Global*

VB

Bisher war es Ihnen unmöglich, einen *System.IO*-Namespace in der Namespace-Hierarchie Ihrer Firma zu definieren ...

```
Namespace MeineFirma.System.IO          ' nicht möglich!
```

... denn dies hätte zu Konflikten mit den Verweisen auf den *System*-Namespace in Ihrer Anwendung geführt.

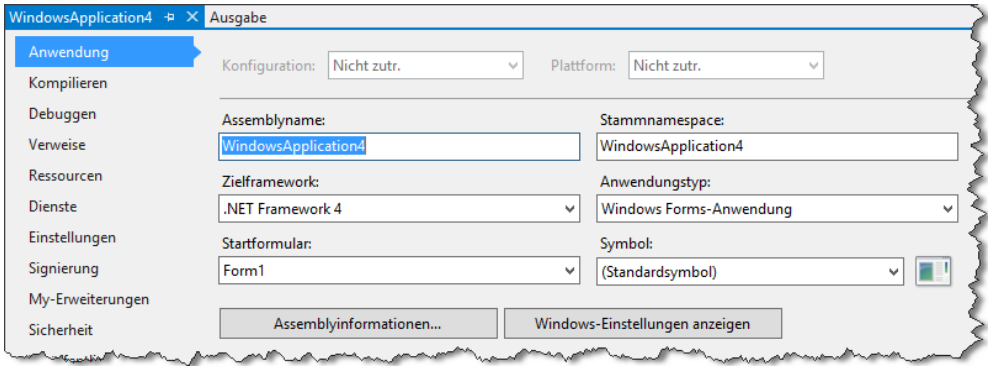
Mit dem Schlüsselwort *Global* können Sie die Namespaces nun unterscheiden:

```
Dim aFile As Global.System.IO.File
```

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.

5.1.6 Eigene Namespaces einrichten

Den von der Entwicklungsumgebung für Ihr neues Projekt automatisch vergebenen Namespace (Stammnamespace) können Sie auf der Seite *Anwendung* des Projekteigenschaften-Dialogs ändern.



Ein neues Windows Forms-Projekt erhält standardmäßig von der IDE den Namen *WindowsApplication1,* falls nicht bereits ein gleichnamiges Projekt existiert. Folglich wird es automatisch in den Stammnamespace *WindowsApplication1* eingebunden.

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 5.5: Zwei verschachtelte Namespaces werden innerhalb des Stammnamespace eingerichtet.

VB
```
Namespace myNamespace1
    Public Class meineKlasse1
        ...
    End Class

    Namespace myNamespace2
        Public Class meineKlasse2
            ...
        End Class
    End Namespace
End Namespace
```

Und vergessen Sie niemals:

HINWEIS: Ein Typ in einem übergeordneten Namespace hat **nicht** automatisch Zugriff auf einen Typ im untergeordneten Namespace!

5.2 Überladen von Operatoren

Das Überladen von Operatoren ermöglicht es, den bereits in VB vorhandenen Standardoperatoren (+, -, ...) eine beliebige andere Bedeutung zuzuweisen.

5.2.1 Syntaxregeln

Die Syntax basiert auf dem Schlüsselwort *Operator.*

SYNTAX:
```
Public Shared Operator Operator (Operand1, Operand2) As Ergebnistyp
    ...
    Return ergebnis
End Operator
```

Obige Syntax gilt für binäre Operatoren (+,-, *, /, ...), die zwei Operanden für die Ausführung der Operation brauchen.

Für unäre Operatoren (+=, ...) gilt:

```
SYNTAX: Public Shared Operator Operator (Operand) As Ergebnistyp
            ...
            Return ergebnis
        End Operator
```

5.2.2 Praktische Anwendung

Wie bereits erwähnt, können nur die in VB bereits vorhandenen Standard-Operatoren überladen werden. Im folgenden Beispiel erhält der "+"-Operator eine neue Bedeutung:

Zwei gleichgroße eindimensionale *Double*-Arrays sollen mittels einer Überladung des "+"-Operators addiert werden. Es folgt eine Beschreibung der dazu notwendigen Klassen.

Klasse CArray

Die Klasse *CArray* kapselt den Zugriff auf ein internes eindimensionales Array:

```
Public Class CArray
    Public Shared N As Integer          ' Anzahl der Elemente
    Private arr() As Double             ' 1-dim. Array
```

Der Konstruktor erzeugt das Array in entsprechender Größe:

```
    Public Sub New(nr As Integer)        ' Konstruktor
        N = nr
        ReDim arr(nr)
    End Sub
```

Ein als Standardeigenschaft getarnter Indexer ermöglicht den bequemen indizierten Zugriff auf die Elemente der Instanzen der Klasse *CArray*:

```
    Default Public Property Index(i As Integer) As Double
        Get
            Return arr(i)
        End Get
        Set(value As Double)
            arr(i) = value
        End Set
    End Property
```

Eine Überladung des "+"-Operators wird definiert. Rückgabewert ist eine neue Instanz von *CArray*:

```
    Public Shared Operator +(a1 As CArray, a2 As CArray) As CArray
        Dim a As New CArray(N)
        For i As Integer = 0 To N - 1
            a(i) = a1(i) + a2(i)
        Next i
        Return a
    End Operator
End Class
```

Klasse Form1

Anwendung der Operatorenüberladung bei der Addition von zwei Instanzen von *CArray*:

```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim A1 As New CArray(3)
        A1(0) = 1.5 : A1(1) = 5.25 : A1(2) = -0.75
        Dim A2 As New CArray(3)
        A2(0) = 0 : A2(1) = 2.3 : A2(2) = -7.1
        Dim A3 As CArray = A1 + A2              ' Anwendung des überladenen "+"-Operators!
        For i As Integer = 0 To CArray.N - 1
            MessageBox.Show(A3(i).ToString)    '  "1,5", "7,55", "-7,85"
        Next i
    End Sub
End Class
```

5.2.3 Konvertierungsoperatoren überladen

Nicht nur "normale" Operatoren, sondern auch Konvertierungsoperatoren wie *CType*, können überladen werden. Hier muss allerdings noch das Schlüsselwort *Widening* oder *Narrowing* angeben werden:

- *Widening* "versichert" dem Compiler, dass die Umwandlung auf jeden Fall fehlerfrei durchgeführt wird und dass kein Informationsverlust eintritt (implizite Typumwandlung, z.B. bei Umwandlungen von *Integer* nach *Long*).
- *Narrowing* bedeutet genau das Gegenteil: hier muss damit gerechnet werden, dass ein Fehler auftritt oder dass Informationen verloren gehen (explizite Typumwandlung).

BEISPIEL 5.6: Konvertierungsoperatoren überladen

VB

(Fortsetzung) Wenn Sie im obigen Beispielcode den Wert einer Instanz von *CArray* mit *CType(A3, String)* in einen String konvertieren wollen, gibt es einem Compilerfehler, denn der Typ *CArray* kann nicht in *String* umgewandelt werden. Durch eine Überladung des *CType*-Konvertierungsoperators können Sie aber ein wunschgemäßes Ergebnis erhalten:

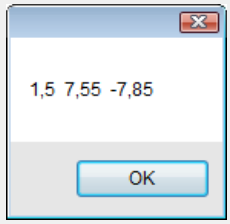
```
Public Shared Widening Operator CType(Value As CArray) As String
    Dim s As String = String.Empty
    For i As Integer = 0 To N - 1
        s &= Value(i) & "  "
    Next
    Return s
End Operator
```

Der Test:

```
MessageBox.Show(CType(A3, String))
```

BEISPIEL 5.6: Konvertierungsoperatoren überladen

Ergebnis



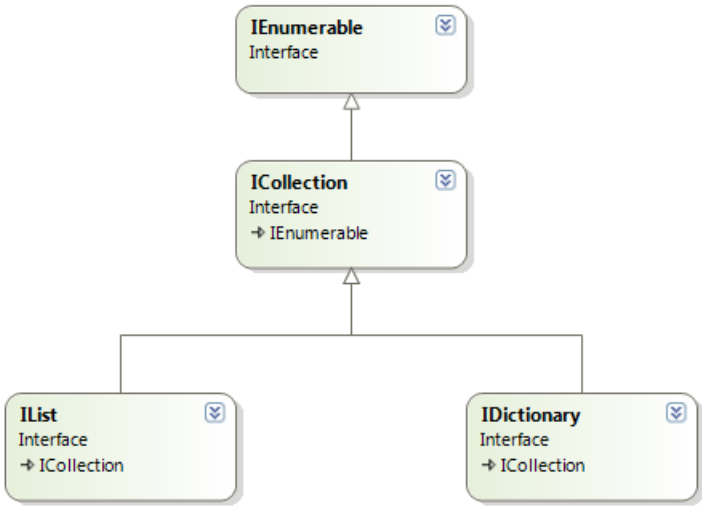
5.3 Auflistungen (Collections)

Wenn Sie mehrere Objekte speichern wollen, so ist es häufig günstiger, dies statt in einem Array 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. Doch zunächst einige vorbereitende Betrachtungen.

5.3.1 Beziehungen zwischen den Schnittstellen

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 spalten 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).

5.3.2 IEnumerable

Diese Schnittstelle stellt die Kernfunktionalität einer Collection bereit und macht einen Enumerator verfügbar, der ein einfaches Iterieren 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 5.7: Der Inhalt einer mit *String*-Objekten gefüllten Auflistung *coll* wird in einer *ListBox* angezeigt.

VB
```
Dim en As IEnumerator = coll.GetEnumerator()              ' Enumerator-Objekt referenzieren
Do While en.MoveNext()
    ListBox1.Items.Add(en.Current)
Loop
```

Wie wichtig der im Hintergrund agierende Enumerator ist, wird am Beispiel der *For Each*-Schleife deutlich, mit welcher eine beliebige Auflistung komplett durchlaufen werden kann.

BEISPIEL 5.8: Das Vorgängerbeispiel wird mit einer *For Each*-Schleife gelöst.

VB
```
For Each str As String in coll
    ListBox1.Items.Add(str)
Next
```

HINWEIS: Verwechseln Sie 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.

5.3.3 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 das Auflistungsobjekt synchronisiert (threadsicher) ist
SyncRoot	... liefert eine Objektreferenz, die den Objektzugriff synchronisiert
CopyTo()	... kopiert die Elemente der Auflistung in ein Array

5.3.4 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.

5.3.5 Iteratoren

Die Schnittstelle *IEnumerator* unterstützt bekanntlich eine Methode mit dem Namen *GetEnumerator* (siehe 5.3.2). Für die Rückgabe der einzelnen Werte kann das Schlüsselwort *Yield* benutzt werden.

BEISPIEL 5.9: Eine Klasse *CMonate* implementiert *GetEnumerator*

VB
```
Private Class CMonate
       Public Iterator Function GetEnumerator() As IEnumerator
           Yield "Januar"
           Yield "Februar"
           Yield "März"
           ...
           Yield "Dezember"
       End Function
End Class
```

Dank *Yield* kann die Klasse kann mittels *For Each*-Schleife durchlaufen werden:

```
Dim monate As New CMonate()
For Each s As String In monate
    ListBox1.Items.Add(s)
Next
```

Wie nicht anders zu erwarten, erscheinen in der *ListBox* untereinander die zwölf Monate.

5.3.6 Die ArrayList-Collection

Die *ArrayList* ist eine nullbasierte Collection, der mittels *Add*-Methode beliebige Elemente hinzugefügt werden können. 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 5.10: Drei Nachkommen der Klasse *CKunde* unterschiedlichen Typs werden in einer *ArrayList* gespeichert:

VB
```
Dim kunde1 As New CPrivatKunde("Herr", "Krause", "Leipzig")
Dim kunde2 As New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
Dim kunde3 As New CFirmenKunde("Herr", "Maus", "Manfreds Internet AG")

Dim kunden As New ArrayList(3)        ' Startkapazität 3
kunden.Add(kunde1)
kunden.Add(kunde2)
kunden.Add(kunde3)
```

Für alle Kunden-Objekte wird die (polymorphe) *addGuthaben*-Methode aufgerufen:

```
For Each ku As CKunde In kunden
     ku.addGuthaben(brutto)
Next ku
```

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 5.11: (Fortsetzung) Das Guthaben des ersten Kunden soll angezeigt werden.

VB

Die folgende Anweisung führt zu einem Konvertierungsfehler:

```
Label1.Text = kunden(0).Guthaben.ToString("C")                ' Fehler!
```

Das aber funktioniert:

```
Dim kd As CKunde =  CType(kunden(0), CKunde)                  ' Typecasting
Label1.Text = kd.Guthaben.ToString("C")
```

Mit *AddRange* fügen Sie Elemente aus einer anderen Auflistung hinzu.

BEISPIEL 5.12: Zwei Personen werden der Kundenliste hinzugefügt.

VB

```
Dim personen As ArrayList = 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.

5.3.7 Die 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 *Integer*- 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 5.13: *Hashtable*

VB

Die drei im Vorgängerbeispiel erzeugten Instanzen der Klassen *CPrivatKunde* bzw. *CFirmenKunde* werden mit den Integer-Schlüsseln 101, 102, 103 in einem *Hashtable*-Objekt gespeichert.

```
Dim kunden As New Hashtable()
kunden.Add(101, kunde1)
```

BEISPIEL 5.13: *Hashtable*

```
kunden.Add(102, kunde2)
kunden.Add(103, kunde3)
```

Für alle Kunden-Objekte wird die *addGuthaben*-Methode aufgerufen:

```
For Each ku As CKunde In kunden.Values
     ku.addGuthaben(brutto)
Next ku
```

Der direkte Zugriff auf das Element mit dem Schlüssel 102:

```
Dim kd As CKunde = CType(kunden(102), CKunde)
Label1.Text = kd.Guthaben.ToString("C")
```

HINWEIS: Weil Sie die Felder einer *Hashtable* nicht normal indizieren können, ist das Durchlaufen mittels *For-Next*-Schleife nicht möglich, Sie müssen also immer die *For Each*-Schleife verwenden.

5.4 Generische Datentypen

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 wie Strukturen, Schnittstellen, Delegaten und 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 Klammern (*Of Datentyp*) unmittelbar nach dem Klassen- bzw. Methodenbezeichner angegeben.

BEISPIEL 5.14: Erzeugen einer generischen Liste zum Speichern von *Integer*-Zahlen

VB
```
Dim gL As New System.Collections.Generic.List(Of Integer)
```

Um die Vorzüge der Generics richtig würdigen zu können, wollen wir uns zunächst noch einmal an frühere Zeiten erinnern (unter .NET 1.x).

5.4.1 Wie es früher einmal war

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, dass die Elemente der *ArrayList* als *System.Object* gespeichert werden.

BEISPIEL 5.15: *CKunde*-Objekte werden in einer *ArrayList* gespeichert und ausgelesen.

VB
```
Public Class CKunde
        Private _nachName As String

        Public Sub New(nN As String, a As Integer)
            Me._nachName = nN
        End Sub

        Public Property NachName() As String
            Get
                Return Me._nachName
            End Get
            Set(value As String)
                _nachName = value
            End Set
        End Property
End Class
```

ArrayList erzeugen und zwei Objekte hinzufügen:

```
Dim aList As 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 i As Integer = 0 To aList.Count - 1
     Dim kd As CKunde = CType(aList(i), CKunde)
     ListBox1.Items.Add(kd.NachName)
Next i
```

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 5.16: Zu unserer obigen Kundenliste lassen sich problemlos auch Zahlen hinzufügen.

VB
```
Dim aList As New System.Collections.ArrayList()
aList.Add(New CKunde("Lehmann"))
aList.Add(44)     ' kein Fehler!
```

Das böse Erwachen kommt beim Zugriff:

```
For Each kd As CKunde In aList
  ListBox1.Items.Add(kd.NachName)      ' Fehler!
Next kd
```

Eine eigene generische Collection-Klasse bedeutet viel Arbeit

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 5.17: Ein *Kunden*-Array wird erzeugt und mit zwei Elementen initialisiert.

VB
```
Dim kunden() As CKunde = New CKunde() {New CKunde("Müller"), New CKunde("Meier")}
For Each kd As CKunde In kunden
       ListBox1.Items.Add(kd.NachName)
Next kd
```

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 Typsicherheit durch Generics

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.

Die Syntax der Deklaration ist ähnlich wie bei der Verwendung der Klasse, nach dem *Of*-Schlüsselwort folgen ein Platzhalter für einen oder mehrere Datentypen. Diese Platzhalter können wir nun dort verwenden, wo wir ansonsten einen Typen verwendet hätten.

BEISPIEL 5.18: Aufgabe der Klasse *CStore(Of T)* ist es, den Wert eines beliebigen Typs *T* zu speichern.

VB
```
Public Class CStore(Of T)
  Dim tp As T
```

BEISPIEL 5.18: Aufgabe der Klasse *CStore(Of T)* ist es, den Wert eines beliebigen Typs *T* zu speichern.

```
  Public Property Val As T
      Get
         Return tp
      End Get
      Set (value As T)
         tp = value
      End Set
  End Property
End Class
```

Der Klassennamen *CStore(Of T)* ist der erste Hinweis, dass es sich um einen generischen Typ handelt, die runden Klammern umschließen das *Of*-Schlüsselwort und den Typen-Platzhalter *T*. Die Variablendeklaration *Dim tp As 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 übergeben.

BEISPIEL 5.19: *(Fortsetzung)* Verwenden der Klasse *CStore* mit verschiedenen Datentypen und Anzeige in einer *ListBox*.

VB

Für *String*-Datentyp:

```
Dim aString As New CStore(Of String)
aString.Val = "Hallo"
ListBox1.Items.Add(aString.Val)
```

Für Datentyp *CKunde*:

```
Dim kunde As New CStore(Of CKunde)
kunde.Val = New CKunde("Müller")
ListBox1.Items.Add(kunde.Val.NachName)
```

Obwohl wir in diesem Beispiel ein und dieselbe Klasse *CStore* verwendet haben um einen String oder einen Kunden zu speichern, bleibt ihr ursprünglicher Typ unverändert. Und das alles funktioniert, ohne dass wir *System.Object* verwenden mussten!

5.4.3 List-Collection ersetzt ArrayList

Das wohl wichtigste Einsatzgebiet für Generics sind Collections jeglicher Art. Aus Gründen der Abwärtskompatibilität musste allerdings auf eine Änderung/Erweiterung der im Namespace

System.Collections bereits vorhandenen Klassen verzichtet werden. Als Alternative zur *ArrayList* bietet sich jetzt die generische *List* Klasse an.

BEISPIEL 5.20: Erzeugen einer generischen List für den benutzerdefinierten Typ *CKunde* (Deklaration der Klasse *CKunde* siehe oben).

VB
```
Dim kunden As New System.Collections.Generic.List(Of CKunde)
```

HINWEIS: Achten Sie darauf, dass der *Generic*-Namespace (innerhalb des *System.Collections* Namespace) eingebunden ist.

Dort werden Sie alle neuen (generischen) Klassen wie *List*, *SortedList,* Dictionary, *Queue*, *Stack*, ... finden. Die *List*-Klasse in diesem Namespace ähnelt der *System.Collections.ArrayList* Klasse, allerdings gibt es einige wichtige Unterschiede.

BEISPIEL 5.21: (*Fortsetzung*)

VB

Um die Verwendung der generischen *List* zu zeigen, wollen wir dreimal eine Schleife durchlaufen in welcher wir einen neuen Kunden (*Müller1*, *Müller2*, *Müller3*) erzeugen und diesen zur Collection hinzufügen:

```
For i As Integer = 0 To 2
      Dim Kunde As New CKunde("Müller" + (i+1).ToString)
      kunden.Add(kunde)
Next i
```

Durch die Collection iterieren und jedes Element in einer *ListBox* ausgeben:

```
For Each kd As CKunde In kunden
     ListBox1.Items.Add(kd.NachName)
Next kd
```

Eine andere Möglichkeit zur Ausgabe der Liste wäre eine einfache *For*-Schleife.

BEISPIEL 5.22: Die Nachnamen aller Kunden in einer *ListBox* anzeigen

VB
```
For i As Integer = 0 To kunden.Count - 1
      ListBox1.Items.Add(kunden(i).NachName)
Next i
```

5.4.4 Über die Vorzüge generischer Collections

Die Arbeit mit einer generischen *List* unterscheidet sich deutlich von unseren durch die *ArrayList* geprägten Gewohnheiten. Wir brauchen uns keinerlei Gedanken mehr über den zu speichernden Datentyp 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 die generische Liste "weiß", welcher Datentyp zu speichern und wie viel Speicher dafür zu reservieren ist.
- Anstatt 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.

Das folgende Beispiel soll unsere Betrachtungen abrunden. Zwar gibt es bereits im Namespace *System.Collection.Generic* eine Klasse *Stack*, wir aber wollen eine generische Collection *CStack* mit ähnlicher Funktionalität selbst programmieren.

BEISPIEL 5.23: Unsere Klasse *CStack* speichert beliebige Datentypen nach dem Push/Pop-Prinzip und benutzt selbst die generische *List*-Klasse (siehe oben).

VB
```
Public Class CStack(Of Type)
      Private _list As New List(Of Type)          ' generische List-Klasse
      Public Function Pop() As Type
            Dim index As Integer = _list.Count - 1
            Dim val As Type = _list(index)
            _list.RemoveAt(index)
            Return val
        End Function

        Public Sub Push(item As Type)
            _list.Add(item)
        End Sub
End Class
```

Der Zugriff am Beispiel des Speicherns von *Integer*-Zahlen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim s1 As New CStack(Of Integer)
        For i As Integer = 1 To 5
            s1.Push(i)
        Next i
        For i As Integer = 1 To 5
            MessageBox.Show(s1.Pop.ToString)     ' zeigt 5, 4, 3, 2, 1
        Next i
End Sub
```

5.4.5 Typbeschränkungen durch Constraints

Mitunter ist es ein Nachteil, dass eine (generische) Klasse beliebige Datentypen verarbeiten kann. Mit so genannten *Constraints* kann man festlegen, welche Typen als Typ-Platzhalter infrage kommen. Man verwendet dazu das der Klassendefinition nachgestellte *As*-Schlüsselwort.

BEISPIEL 5.24: Typbeschränkung

VB

Die Klasse *CStack* kann nur noch Objekte (Referenztypen) speichern, die das *IComparable*-Interface unterstützen. Ein Speichern von Integer-Zahlen ist also nicht mehr möglich.

```
Class CStack(Of Type As {Class, IComparable})
```

BEISPIEL 5.25: Der generischen Typ *Dictionary* mit einem *Constraint*

VB

```
Public Class Dictionary(Of T As IComparable)
```

Die *As*-Klausel beschränkt den *T*-Typ auf den Typ *IComparable*. Falls *T* das *IComparable*-Interface nicht implementiert, gibt es einen Compiler-Fehler.

5.4.6 Collection-Initialisierer

Collection-Initialisierer ermöglichen eine verkürzte Syntax beim Erzeugen von Auflistungen, die mit vorhandenen Werten gefüllt werden sollen.

Ein Collection-Initialisierer besteht aus einer kommaseparierten Werteliste, die in geschweiften Klammern einzuschließen ist und der das Schlüsselwort *From* vorangestellt wird.

BEISPIEL 5.26: Eine einfache Namensliste

VB

```
Dim names As New List(Of String) From {"Günther", "Erhard", "Michael"}
```

BEISPIEL 5.27: Eine Liste für Menüeinträge

VB

```
Dim dateiMenü = New List(Of MenuOption) From {{1, "Neu"}, {2, "Öffnen"},
                                              {3, "Speichern"}, {4, "Beenden"}}
```

Wenn Sie eine generische Collection wie *List(Of T)* oder *Dictionary(Of TKey, TValue)* erzeugen wollen, so müssen Sie den Typ der Collection vor dem Collection-Initializer deklarieren.

BEISPIEL 5.28: Deklaration einer generischen Collection als Eigenschaft

VB

```
Public Property Items As List(Of String) = New List(Of String) From {"Eier", "Butter",
                                                                      "Käse"}
```

HINWEIS: Collection-Initialisierer lassen sich nicht mit Objekt-Initialisierern (siehe 3.8.2) kombinieren!

5.4.7 Generische Methoden

Eine Methode mit Typ-Parameter ist generisch (*Generic Method*). Ähnlich wie bei der Klasse wird der Typ in runden Klammern (mit vorangestelltem *Of*-Schlüsselwort) angegeben.

BEISPIEL 5.29: Generische Methode für Vertauschen von zwei beliebigen Objekten

VB
```
Private Sub genSwap(Of T)(ByRef a As T, ByRef b As T)
      Dim tmp  As T = a
      a = b
      b = tmp
End Sub
```

Anwendung für *Image*-und für *String*-Objekte:

```
Dim im1, im2 As Image
Dim s1, s2 As String
...
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    genSwap(im1, im2)
    PictureBox1.Image = im1
    PictureBox2.Image = im2
    genSwap(s1, s2)
    Label1.Text = s1
    Label2.Text = s2
End Sub
```

5.5 Delegates

Die Delegate-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.

5.5.1 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).

Das Arbeiten mit einem Delegate unterscheidet sich prinzipielle nicht von der gewohnten OOP-Strategie (Klasse bzw. Typ definieren, Objektvariable referenzieren, erzeugen und initialisieren). Insgesamt sind folgende Schritte erforderlich:

1. Definition des Delegattypen mittels *Delegate*-Schlüsselwort.
2. Deklaration eines Objekts vom Typ des Delegaten.
3. Instanziieren der Delegaten mittels *New* (erst hier wird das Delegate-Objekt erzeugt, welches auf die konkrete Methode zeigt. Als Argument wird die Adresse der Methode übergeben, die vom Delegaten aufgerufen werden soll.
4. Der Aufruf des Delegate-Objekts (an die Stelle der konkreten Methode tritt das Delegate-Objekt).

Die Schritte im Einzelnen:

5.5.2 Delegate-Typ definieren

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** Function *Name*([*<Parameterliste>*]) As *Rückgabetyp*

SYNTAX: *<Sichtbarkeit>* **Delegate** Sub *Name*([*<Parameterliste>*])

HINWEIS: Die Parameterliste einer Delegate-Typdefinition entspricht der Parameterliste der Funktionen/Prozeduren, auf die die daraus erzeugten Delegate-Objekte zeigen sollen.

BEISPIEL 5.30: Eine Delegate für eine Funktion *compA*, die zwei *Integer*-Zahlen vergleicht

VB
```
Private Function compA(x As Integer, y As Integer) As Boolean
    If y < x Then
        Return True
    Else
        Return False
    End If
End Function
```

Delegate-Typ für obige Funktion definieren:

```
Private Delegate Function CompDlg(x As Integer, y As Integer) As Boolean
```

5.5.3 Delegate-Objekt erzeugen

Ein Delegate kann entweder mit einer benannten Methode oder aber auch mit einer anonymen Methode oder einem Lambda Expression instanziiert werden.

BEISPIEL 5.31: *(Fortsetzung)* Delegate-Objekt des Typs *CompDlg* für die Methode *compA* erzeugen.

VB
```
Dim compMethod As CompDlg = Nothing                  ' referenzieren
compMethod = New CompDlg(AddressOf compA)            ' instanziieren
```

Mit dem Delegate-Objekt arbeiten

Die Verwendung unseres Delegate-Objekts unterscheidet sich nicht von der Verwendung einer herkömmlichen benannten Methode.

BEISPIEL 5.32: *(Fortsetzung)* Funktionsaufruf des Delegate-Objekts *compMethod*

VB
```
MessageBox.Show(compMethod(5, 4).ToString)              ' zeigt "True"
```

Zum Vergleich der Aufruf der benannten Methode *compA*:

```
MessageBox.Show(compA(5, 4).ToString)                   ' zeigt "True"
```

5.5.4 Delegates vereinfacht instanziieren

Wie auch bei normalen Objekten üblich, können auch bei Delegates das Referenzieren und das Instanziieren in einer einzigen Anweisung realisiert werden.

BEISPIEL 5.33: Vereinfachtes Referenzieren und Instanziieren des Delegate-Objekts aus dem Vorgängerbeispiel

VB
```
Dim compMethod As New CompDlg(AddressOf compA)
```

Auch auf den *New*-Operator kann verzichtet werden!

BEISPIEL 5.34: *(Fortsetzung)* Das Vorgängerbeispiel lässt sich weiter vereinfachen.

VB
```
Dim compMethod As CompDlg = AddressOf compA
```

Da das Delegate-Konzept auch für Generics gilt, sorgt auch hier die vereinfachte Instanziierung für mehr Übersicht.

BEISPIEL 5.35: Obiges Beispiel mit einem generischen Delegaten für den Vergleich von *Double*-Zahlen

VB
```
Private Delegate Function CompDlg(Of T)(x As Integer, y As Integer) As Boolean
Dim compMethod As CompDlg(Of Double) = AddressOf compA
...
```

5.5.5 Relaxed Delegates

Seit VB 9.0 gibt es ein weniger streng typisiertes Delegatenkonzept. In den Vorgängerversionen mussten beim Binden einer Methode an einen Delegaten die Signaturen von Methode und Delegate exakt übereinstimmen. Diese Einschränkung wurde aufgehoben (Relaxed Delegates).

BEISPIEL 5.36: Relaxed Delegates

VB

Zu einem Formular soll per Code ein *Button* zum Schließen des Formulars hinzugefügt werden. Dabei musste bislang die Signatur der *OnClick*-Ereignisbehandlungsroutine exakt mit der des (im Hintergrund deklarierten) Eventhandler-Delegaten der *Button*-Klasse übereinstimmen:

```
Delegate Sub EventHandler(sender As Object, e As EventArgs)
```

Zunächst der klassische Code:

```
Private WithEvents btn As New Button

Private Sub OnClick1(sender As Object, e As EventArgs) Handles btn.Click
        Me.Close()
End Sub

Private Sub Form1_Load(sender As Object, e As EventArgs) Handles Me.Load
        btn.Text = "Beenden"
        Me.Controls.Add(btn)
End Sub
```

Ab VB 9.0 lässt sich derselbe Code wie folgt vereinfachen:

```
Private WithEvents btn As New Button

Private Sub OnClick1() Handles btn.Click
        Me.Close()
End Sub

Private Sub Form1_Load() Handles Me.Load
        btn.Text = "Beenden"
        Controls.Add(btn)
End Sub
```

5.5.6 Anonyme Methoden

Bei komplexeren Programmen kann der Aufruf benannter Methoden 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 5.37: Anonyme Methoden für Addition und Multiplikation zweier *Integer*-Zahlen.

VB
```
Delegate Function opIntDlg(x As Integer, y As Integer) As Integer
```

Instanziierung mittels anonymer Methoden:

```
Dim addMethod As opIntDlg = Function(x As Integer, y As Integer)
                              Return x + y
                            End Function

Dim multMethod As opIntDlg = Function(x As Integer, y As Integer)
                               Return x * y
                             End Function
```

Beispiele für Aufrufe:

```
Label1.Text = addMethod(4, 7).ToString                    ' zeigt 11
Label2.Text = multMethod(4, 7).ToString                   ' zeigt 28
```

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.

5.5.7 Lambda-Ausdrücke

Eng verwandt mit den anonymen Methoden sind die so genannten Lambda-Ausdrücke. Die Syntax basiert auf dem Schlüsselwort *Function*.

SYNTAX: `Function(Inputparameter) Expression`

Die *Inputparameter* werden im *Expression* ausgewertet und liefern so gewissermaßen den Rückgabewert des Lambda-Ausdrucks[1].

Ein Lambda-Ausdruck ist quasi eine namenlose Funktion, von der ein einzelner Wert berechnet und zurückgegeben wird. Im Gegensatz zu benannten Funktionen kann ein Lambda-Ausdruck gleichzeitig definiert und ausgeführt werden. Im folgenden Beispiel wird *True* angezeigt.

BEISPIEL 5.38: Das Vorgängerbeispiel wird anstatt mit anonymen Methoden mit Lambda-Ausdrücken realisiert.

VB
```
Delegate Function opIntDlg(x As Integer, y As Integer) As Integer
```

[1] Seit VB 10 dürfen sich Lambda-Ausdrücke auch über mehrere Zeilen erstrecken und aus beliebigen Functions/Subs bestehen. Im alten Visual Basic 8 war der Umgang mit Lambdas eingeschränkt, da diese nur einzeilig sein durften und grundsätzlich Funktionsergebnisse zurückliefern mussten.

BEISPIEL 5.38: Das Vorgängerbeispiel wird anstatt mit anonymen Methoden mit Lambda-Ausdrücken realisiert.

Instanziierung mittels Lambda Ausdrücken:

```
Dim addMethod As opIntDlg = Function(x As Integer, y As Integer) x + y
Dim multMethod As opIntDlg = Function(x As Integer, y As Integer) x * y
...
MessageBox.Show(addMethod(5, 4).ToString)        ' zeigt 9
...
```

Lambda-Ausdrücke haben ihr Haupteinsatzgebiet innerhalb der LINQ-Technologie (siehe Kapitel 6). Einen kleinen Vorgeschmack liefert das folgende Beispiel.

BEISPIEL 5.39: Definition einer Abfrage über eine (generische) Liste.

VB
```
Dim employees As List(Of Employee) = New List(Of Employee)
Dim query = employees.FindAll(Function(c) c.City = "London")
```

HINWEIS: In LINQ liegen vielen der Standardabfrageoperatoren Lambda-Ausdrücke zugrunde, diese werden vom Compiler erstellt, um Berechnungen zu erfassen, die in grundlegenden Abfragemethoden wie *Where*, *Select*, *Order By*, *TakeWhile* usw. definiert sind.

5.5.8 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.

HINWEIS: Ausführlich gehen wir auf die *Task Parallel Library* und die parallele Programmierung erst im Kapitel 13 ein.

Um Ihnen dennoch einen ersten Eindruck zu vermitteln, sollen zwei Beispiele gegenübergestellt werden.

BEISPIEL 5.40: Eine einfache *For*-Schleife in klassischer Programmiertechnik

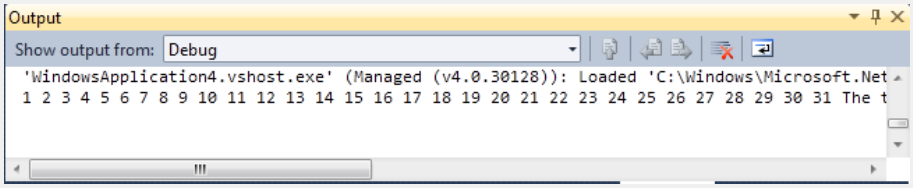
VB
```
        Dim N As Integer = 100
        For i As Integer = 1 To N - 1
            Debug.Write(i.ToString & " ")
            If i > 30 Then
                Exit For
```

BEISPIEL 5.40: Eine einfache *For*-Schleife in klassischer Programmiertechnik

```
            End If
        Next i
```

Ergebnis

Die Ausgabe (Menü *Ansicht|Ausgabe*):



Das analoge "Problem" soll nun mittels Parallelprogrammierung gelöst werden.

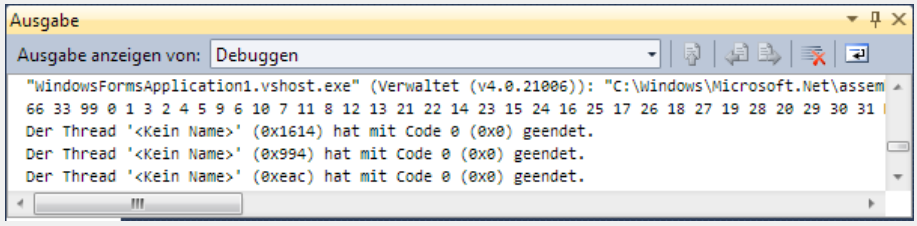
BEISPIEL 5.41: Anwendung eines Lambda-Ausdrucks in der *Parallel.For*-Methode

VB

```
Imports System.Threading.Tasks
...
    Private N As Integer = 100
    Parallel.For(0, N, Sub(i, lState)
                           Debug.Write(i.ToString & " ")
                           If (i > 50) Then lState.Break()
                       End Sub)
...
```

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.

5.6 Dynamische Programmierung

Sie gehörte zweifelsfrei zu den Highlights von .NET 4.0 – die so genannte *dynamische* Programmierung. 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 Visual Basic jetzt die Ausführung beliebiger Operationen mit Variablen eines dynamischen Typs bzw. bei der Definition eigener Typen. Doch ehe wir ins Detail gehen, sollen zunächst einige allgemeine Dinge geklärt werden.

5.6.1 Wozu dynamische Programmierung?

Die meisten VB Entwickler verwenden mehrere Werkzeuge in einer einzigen Applikation, um komplexe Aufgaben zu programmieren. Für WPF Desktop-Anwendungen braucht man beispielsweise VB und XAML. Häufig findet sich Open-Source-Code zum Lösen eines Problems, allerdings kann dieser in einer anderen Sprache vorliegen, z.B. in C# 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.
- Dynamische Objekte gestatten einen bequemen Zugriff auf dynamische Sprachen wie IronPython und IronRuby. Sie können beispielsweise ein dynamisches Objekt verwenden, um ein dynamisches Skript zu referenzieren, welches erst zur Laufzeit interpretiert wird.

5.6.2 Das Prinzip der dynamischen Programmierung

Dynamische Objekte entscheiden über ihre Eigenschaften und Methoden erst zur Laufzeit anstatt bereits beim Kompilieren. Der Programmierer hat dadurch die Möglichkeit, mit Objekten zu arbeiten, deren Typ vorher nicht festliegt und erst zur Laufzeit interpretiert wird.

HINWEIS: Ein dynamisches Objekt wird wird durch spätes Binden referenziert, unter Visual Basic wird der Typ als *Object* spezifiziert[1].

Die Klasse DynamicObject

Für die dynamische Programmierung ist der Namespace *System.Dynamic* einzubinden, über welchen die abstrakte Klasse *DynamicObjec*t verfügbar wird. Um Runtime-Funktionalität bereitzustellen müssen Sie einige Methoden dieser Klasse überschreiben (siehe folgende Tabelle).

Methode	Wird aufgerufen wenn ...
TryBinaryOperation	... binäre Operationen, wie Addition oder Multiplikation, ausgeführt werden sollen.
TryConvert	... Typkonvertierungen vorgenommen werden sollen.
TryCreateInstance	... der Typ initialisiert wird.
TryGetIndex	... ein Wert per Index abgerufen werden soll (Indexer).
TryGetMember	... der Wert einer Eigenschaft abgerufen werden soll.
TryInvokeMember	... wenn eine Methode aufgerufen werden soll.
TrySetIndex	... ein Wert per Index gesetzt werden soll (Indexer).
TrySetMember	... der Wert einer Eigenschaft gesetzt werden soll.

Auskunft über die vorhandenen (dynamischen) Member gibt Ihnen die Enumeration *GetDynamicMemberNames*.

Ein kleines Beispiel soll für einen ersten Einblick genügen.

BEISPIEL 5.42: Eine dynamische Klasse *CDynFile* wird deklariert. In der Klasse wird die von *DynamicObject* geerbte Methode *TryInvokeMember* überschrieben.

VB
```
Imports System.Dynamic

Public Class CDynFile
    Inherits DynamicObject
    ...
    Public Overrides Function TryInvokeMember(binder As InvokeMemberBinder,
                                args() As Object, ByRef result As Object) As Boolean
        Dim trim = True
        Try
            If args.Length > 0 Then trim = Convert.ToBoolean(args(0))
        Catch
            MessageBox.Show("Trim-Argument muss Boolescher Wert sein!")
        End Try
        result = getPropValue(binder.Name, trim)
        Return If(result Is Nothing, False, True)
```

[1] In C# gibt es dafür den Datentyp *dynamic*.

BEISPIEL 5.42: Eine dynamische Klasse *CDynFile* wird deklariert. In der Klasse wird die von *DynamicObject* geerbte Methode *TryInvokeMember* überschrieben.

```
    End Function
End Class
```

Die *TryInvokeMember*-Methode der Klasse *DynamicObject* wird aufgerufen, wenn ein Mitglied der daraus abgeleiteten dynamischen Klasse angefordert wird und dabei Argumente übergeben werden sollen. *True* wird zurückgegeben, wenn das referenzierte Mitglied vorhanden ist, ansonsten *False*.

Die Parameter:

- *binder*
  ... liefert Infos über das referenzierte Mitglied, z.B. dessen Name,
- *result*
  ... übergibt das Resultat der Abfrage des spezifizierten Mitglieds per Referenz,
- *args*
  ... beinhaltet ein Array von Argumenten, die bei Aufruf an das dynamische Mitglied übergeben werden.

Wer bis jetzt nur "Bahnhof" verstanden hat – keine Panik:

HINWEIS: Den kompletten Code finden Sie im Praxisbeispiel 5.8.3!

Die Klasse ExpandoObject

Hierbei handelt es sich um eine Klasse, deren Instanzen Sie zur Laufzeit fast beliebige Eigenschaften, Methoden (per Lambda-Ausdruck) und Ereignisse hinzufügen können. Diese lassen sich nach der Definition natürlich auch dynamisch aufrufen bzw. abfragen. Last, but not least: die definierten Member lassen sich auch löschen.

HINWEIS: Die Klasse unterstützt eine ganze Reihe von Erweiterungsmethoden, mit denen Sie den Funktionsumfang der Klasse beträchtlich erweitern können.

Auch hier soll ein kleines Beispiel für mehr Klarheit sorgen.

BEISPIEL 5.43: Verwendung von *ExpandoObject*

VB

Den Namespace nicht vergessen:

```
Imports System.Dynamic
...
```

Zunächst eine Instanz erzeugen:

```
        Dim myExO As Object = New ExpandoObject()
```

BEISPIEL 5.43: Verwendung von *ExpandoObject*

Wir erzeugen dynamisch einige Eigenschaften mit frei gewählten Namen:

```
myExO.ErsteEigenschaft = "Hallo User"
myExO.ZweiteEigenschaft = 4711
myExO.xyzEigenschaft = Me
```

Wir definieren ein neues Ereignis, dazu muss dieses zunächst erzeugt werden:

```
myExO.EinEreignis = Nothing
```

Nachfolgend können wir (etwas umständlich) den Ereignishandler zuweisen:

```
Dim myhandler As EventHandler = AddressOf MeinEreignisHandler
myExO.EinEreignis = Delegate.Combine(myExO.EinEreignis, myhandler)
```

Abschließend wollen wir noch eine Methode per Lambda-Ausdruck definieren:

```
myExO.EineMethode = New Action(Sub()
                        MessageBox.Show("Hallo User! Es ist " &
                                        System.DateTime.Now.ToShortTimeString)
                         myExO.EinEreignis.Invoke(myExO, New EventArgs())
                    End Sub)
```

Wie Sie sehen, löst unsere neue Methode auch gleich noch unser Ereignis aus!

Das Abfragen der Eigenschaften zu einem späteren Zeitpunkt:

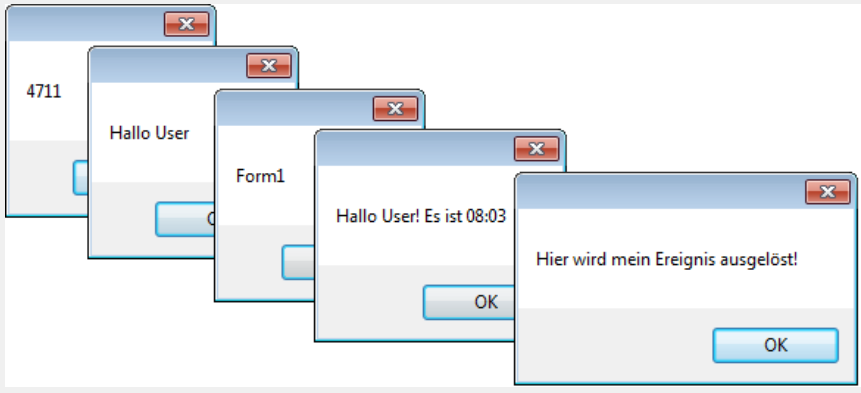
```
MessageBox.Show(myExO.ZweiteEigenschaft)
MessageBox.Show(myExO.ErsteEigenschaft)
MessageBox.Show(myExO.xyzEigenschaft.Text)   ' -> entspricht Me.Text
```

Eine Methode müssen Sie in VB leider mit *Invoke* aufrufen[1]:

```
myExO.EineMethode.Invoke()
```

Ergebnis

Zur Laufzeit dürften die Messageboxen in folgender Reihenfolge erscheinen:



[1] Hier haben es C#-Programmierer einfacher, diese können die Methode direkt aufrufen. Auch die Definition der Ereignisse ist in C# etwas eleganter gelöst.

Late Binding

Das Binden eines dynamischen Typs erfolgt erst zur Laufzeit (spätes Binden) unter Verwendung des *IDynamicMetaObjectProvider* Interface. Die Klassen *DynamicObject* und *ExpandoObject* implementieren dieses Interface. Falls das schiefgeht, benutzt VB stattdessen den klassischen späten Bindungsmechanismus der VB-Runtime.

HINWEIS: Um spätes Binden zu ermöglichen, muss die strenge Typprüfung abgeschaltet werden (*Option Strict Off*).

5.6.3 Kovarianz und Kontravarianz

Das Füllen eines allgemeineren ("breiteren") Typs mit den Werten eines tiefer abgeleiteten ("schmaleren") Typs bezeichnet man als *Kovarianz*. Im umgekehrten Fall, also wenn Sie einem "schmaleren" Typ einen "breiteren" Typ zuweisen, spricht man von *Kontravarianz*.

In Visual Basic erlauben Kovarianz und Kontravarianz eine implizite Konvertierung von Referenzen für Array- und Delegate-Typen sowie für generische Typargumente.

BEISPIEL 5.44: Einem Objektarray wird ein Stringarray zugewiesen und umgekehrt.

VB
```
Option Strict On
...
Dim strings1() = New String {"Günther", "Erhard", "Michael"}
```
Kovarianz:
```
Dim objects() As Object = strings1
```
Kontravarianz:
```
Dim strings2() = CType(objects, String)
MessageBox.Show(strings2(1))      ' "Erhard"
```

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* anstatt eines Arrays, um so Kovarianz und Kontravarianz zu demonstrieren.

BEISPIEL 5.45: (Fortsetzung) Kovarianz und Kontravarianz mit Collections

VB
```
 ...
Dim strings3 As IEnumerable(Of String) = strings1
```
Kovarianz:
```
Dim objects3 As IEnumerable(Of Object) = strings3
```

BEISPIEL 5.45: (Fortsetzung) Kovarianz und Kontravarianz mit Collections

Kontravarianz:

```
Dim strings4 As IEnumerable(Of String) = CType(objects3, IEnumerable(Of String))
MessageBox.Show(strings4(2))     ' "Michael"
```

HINWEIS: Ab .NET 4.0 unterstützen alle Collection-Typen Kovarianz und Kontravarianz.

5.7 Weitere Datentypen

Die folgenden Datentypen werden erst ab der Version 4.0 des .NET-Frameworks unterstützt.

5.7.1 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 5.46: Erzeugen eines *BigInteger* aus einem *long*-Wert

VB
```
Imports System.Numerics
...
Dim bi1 As New BigInteger(1234566)
```

BEISPIEL 5.47: (Fortsetzung) Erzeugen eines *BigInteger* aus einer Zeichenkette

VB
```
Dim bi2 = BigInteger.Parse("12345678901234567890123456789012345678901123456")
```

Auch die mathematischen Operatoren für Addition, Multiplikation, Subtraktion und Division wurden für diese Klasse überladen.

BEISPIEL 5.48: (Fortsetzung) Multiplikation zweier *BigInteger*

VB
```
Dim bi3 As BigInteger = 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 5.49: Es wird bewiesen, dass die Zahl 2097152 eine Potenz von 2 ist

VB
```
Dim bi4 As New BigInteger(2097152)
If bi4.IsPowerOfTwo Then                          ' liefert True
        MessageBox.Show(bi4.ToString & " ist eine Potenz von 2 !")
End If
```

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.
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 5.50: Alternative Durchführung des Vorgängerbeispiels (Multiplikation zweier *BigInteger*-Zahlen)

VB
```
Dim bi3 = BigInteger.Multiply(bi1, bi2)
```

BEISPIEL 5.51: Die ganzzahlige Division *1000000/300000* mit zusätzlicher Ermittlung des Rests

VB
```
Dim bi1 As New BigInteger(1000000)
Dim bi2 As New BigInteger(300000)
Dim bi3 As New BigInteger()

Dim bi4 = BigInteger.DivRem(bi1, bi2, bi3)
```

bi3 hat jetzt den Wert *100000* und *bi4* den Wert *3*

BEISPIEL 5.52: Die 20.-te Potenz von 10 wird berechnet

VB
```
Dim bi1 As New BigInteger(10)
Dim bi2 = BigInteger.Pow(bi1, 20)
```

bi2 hat jetzt einen schier unendlich großen Wert von 100000000000000000000

5.7.2 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 5.53: Addition von zwei komplexen Zahlen

VB

Zwei Instanzen von *Complex* erzeugen:

```
Dim c1 As New Complex(1.5, -0.3)
Dim c2 As New Complex(-5, 2.1)
```

Operation durchführen:

```
Dim c3 As Complex = c1 + c2
```

Ergebnis auswerten

in kartesischen Koordinaten:

```
Dim re As Double = c3.Real        ' -3,5
Dim im As Double = c3.Imaginary   ' 1,8
```

und in Polarkoordinaten:

```
Dim m As Double = c3.Magnitude    ' 3,9357 ...
Dim p As Double = c3.Phase        ' -0,475 ...
```

[1] Eine komplexe Zahl liegt in der Form $a + bj$ vor, dabei ist a der Real- und b der Imaginärteil.

Den mathematisch interessierten Leser wird es freuen, dass die Klasse *Complex* auch zahlreiche statische Methoden (*Cos*, *Cosh*, *Exp* ...) bereitstellt, mit denen eine Vielzahl weiterer Berechnungen durchführbar ist.

5.7.3 Tuple(Of T)

Die generische *System.Tuple(Of T)*-Klasse ermöglicht es, bis zu acht Elemente unterschiedlichen Typs gemeinsam in einem Tupel typsicher abzuspeichern. Mittels der Eigenschaften *Item1 ... Item8* kann dann auf die einzelnen Elemente zugegriffen werden.

BEISPIEL 5.54: Anwendung der *Tuple*-Klasse

VB

Zwei einfache Klassen definieren:

```
Public Class CLektor
    Public Property Name As String
End Class

Public Class CAutor
   Public Property Name As String
End Class
```

Alle Objekte erzeugen (mittels Objektinitialisierer):

```
Dim le1 As New CLektor With {.Name = "Fernando"}
Dim au1 As New CAutor With {.Name = "Walter"}
Dim au2 As New CAutor With {.Name = "Thomas"}
```

Jetzt wird ein *Tuple*-Objekt erzeugt:

```
Dim tuple1 As New Tuple(Of CLektor,CAutor, CAutor) (le1, au1, au2)
```

Der Zugriff erfolgt über die entsprechenden *Item*-Eigenschaften:

```
Dim le As CLektor = tuple1.Item1
MessageBox.Show(le.Name)                    ' zeigt "Fernando"
```

oder

```
Dim au As CAutor = tuple1.Item3
MessageBox.Show(au.Name)                    ' zeigt "Thomas"
```

HINWEIS: Die Elemente eines Tupels sind schreibgeschützt, können also zur Laufzeit nicht verändert werden.

5.7.4 SortedSet(Of T)

Mit der generischen *System.Collections.Generic.SortedSet(Of T)*-Klasse lassen sich Mengen sortieren. Das Hinzufügen und Einsortieren geschieht mittels *Add*-Methode.

BEISPIEL 5.55: Erstellen einer Namensliste und Ausgabe in einer *ListBox*

VB

Eine Instanz erzeugen:

```
Dim Namen As New SortedSet(Of String)()
```

Elemente hinzufügen:

```
Namen.Add("Fernando")
Namen.Add("Thomas")
Namen.Add("Max Walter")
```

Ausgabe:

```
For Each s As String In Namen
    ListBox1.Items.Add(s)
Next
```

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(Of T)* übergeben werden, mit welchem Sie selbst die Sortierfolge bestimmen können.

BEISPIEL 5.56: Die Sortierung soll nach der Zeichenkettenlänge erfolgen

VB

Diese Klasse definiert die neue Sortierung:

```
Public Class CSortierfolge
    Implements IComparer(Of String)

     Public Function Compare(a As String, b As String) As Integer _
                                                 Implements IComparer(Of String).Compare
         Return a.Length - b.Length
    End Function
End Class
```

Die Instanz der sortieren Menge wird nun so erstellt:

```
Dim Namen As New SortedSet(Of String) (New CSortierfolge())
...
```

Ergebnis

Im Vergleich zum Vorgängerbeispiel sieht die Ausgabe jetzt so aus:

```
Thomas
Fernando
Max Walter
```

5.8 Praxisbeispiele

5.8.1 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 5.3.6).

Generics hingegen sind Klassen bzw. Methoden, denen die Typinformationen erst zur Laufzeit zugewiesen werden. Die variablen Datentypen werden 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 5.4.3).

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)

```
Public Class Form1
    Private aL As ArrayList = Nothing
```

Nach Programmstart wird die *ArrayList* erzeugt, mit Werten gefüllt und angezeigt. Den entsprechenden Code fügen wir in den Konstruktor des Formulars ein:

```
    Public Sub New()
        InitializeComponent()
```

Ab hier fügen Sie den Code zum Instanziieren und Initialisieren der *ArrayList* ein:

```
        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()
    End Sub
```

Die Routine zur Anzeige des Inhalts in der *ListBox*:

```
    Private Sub listeAnzeigen()
        ListBox1.Items.Clear()
        For Each str As String In aL
            ListBox1.Items.Add(str)
        Next str
    End Sub
```

Aufsteigend sortieren:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs)  Handles Button2.Click
        aL.Sort()
        listeAnzeigen()
    End Sub
```

Absteigend sortieren:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        aL.Reverse()
        listeAnzeigen()
    End Sub
```

Suchen:

```
    Private ix As Integer = -1                        ' Index des letztmaligen Vorkommens

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim i As Integer = aL.IndexOf(TextBox1.Text, ix + 1)    ' Suche wird ab ix fortgesetzt
        If i < 0 Then
            MessageBox.Show("Das Element wurde nicht (mehr) gefunden!")
            ix = -1
            ListBox1.SelectedIndex = -1      ' Zeilenmarkierung entfernen
        Else
            ListBox1.SelectedIndex = i       ' gefundenes Wort markieren
            ix = i
        End If
    End Sub
End Class
```

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.

```
Public Class Form1
    Private gL As List(Of String) = Nothing

    Public Sub New()
        InitializeComponent()
        gL = New List(Of String)()          ' Instanziierung der generischen Liste
        gL.Add("Das")
        gL.Add("ist")
        gL.Add("ein")
        gL.Add("Test")
        gL.Add("mit")
        gL.Add("der")
        gL.Add("ArrayList.")
        gL.Add("Das")
```

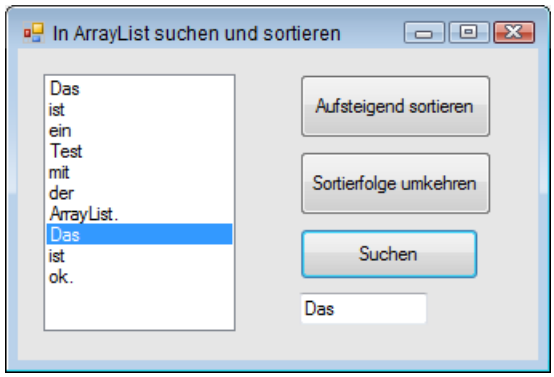
```
        gL.Add("ist")
        gL.Add("ok.")
        listeAnzeigen()
    End Sub

    Private Sub listeAnzeigen()
        ListBox1.Items.Clear()
        For Each str As String In gL
            ListBox1.Items.Add(str)
        Next str
    End Sub
    ...
```

Auch der restliche Code ist analog zum Vorgängerbeispiel.

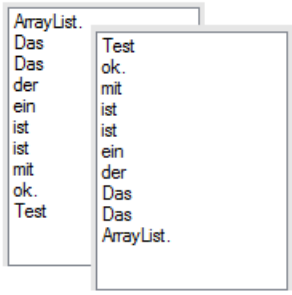
```
    ...
End Class
```

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:

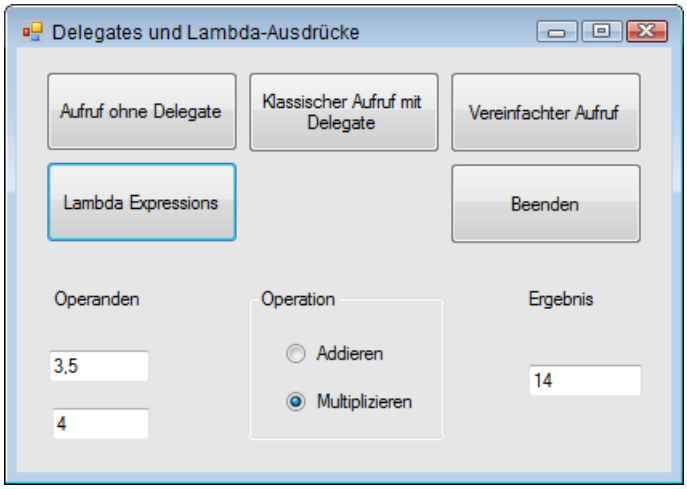
- Das Boxing von Wertetypen ist nicht mehr erforderlich, weil *List(Of String)* in unserem Beispiel jetzt "weiß", welcher Datentyp zu speichern ist und wie viel Speicher man dafür reservieren muss.
- Statt eines Laufzeitfehlers erhalten Sie bereits einen Compiler-Fehler, falls Sie einen falschen Datentyp in der List speichern wollen.
- Eine Typumwandlung (Typecasting) entfällt, wenn Sie auf die Werte der generischen List zugreifen wollen.

5.8.2 Delegates und Lambda Expressions

Dieses Beispiel will dem Einsteiger den Weg zu den etwas schwierigeren Sprachelementen von Visual Basic 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 vier verschiedene und voneinander unabhängige Arten realisiert, wodurch sich gute Vergleichsmöglichkeiten ergeben, um das Wesen von Delegaten (Methodenzeiger) und Lambda Expressions zu erfassen.

Oberfläche

Ein Formular mit fünf *Button*s, zwei in einer *GroupBox* befindlichen RadioButtons und drei TextBoxen bildet unsere Testoberfläche (siehe folgende Laufzeitabbildung).



Über jede der Schaltflächen (außer der zum Beenden) werden die Rechenoperationen auf unterschiedliche Weise ausgeführt.

Quellcode

```
Public Class Form1
```

Zunächst deklarieren wir den Delegaten-Typ *opDeleg*, welcher der Signatur der beiden Methoden *add()* und *mult()* entsprechen muss, die wir danach anschließend deklarieren:

```
    Public Delegate Function opDeleg(x As Single, y As Single) As Double
    Private Function add(x As Single, y As Single) As Double
        Return x + y
    End Function

    Private Function mult(x As Single, y As Single) As Double
        Return x * y
    End Function
```

Gewissermaßen als "Aufwärmübung" werden über den *Button* "Aufruf ohne Delegate" die Rechenoperationen zunächst auf herkömmliche Weise (also ohne Verwendung von Delegates) ausgeführt.

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim a As Single = Convert.ToSingle(TextBox1.Text)
        Dim b As Single = Convert.ToSingle(TextBox2.Text)
        Dim res As Double = 0
        If RadioButton1.Checked Then
            res = add(a, b)
        Else
            res = mult(a, b)
        End If
        TextBox3.Text = res.ToString
    End Sub
```

Nun wollen wir zeigen, wie dieselben Operationen unter Verwendung von Delegate-Objekten realisiert werden können:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Der erste Schritt ist das Deklarieren einer Variablen des eingangs definierten Delegatentyps *opDeleg* (das Prinzip entspricht dem Deklarieren einer Objektvariablen für die Klasse *opDeleg*):

```
        Dim myDlg As opDeleg = Nothing
```

In Abhängigkeit von der eingestellten Rechenoperation wird das Delegate-Objekt *myDlg* entweder mit der *add()*- oder mit der *mult()*-Methode erzeugt:

```
        If RadioButton1.Checked Then
            myDlg = New opDeleg(AddressOf add)
        Else
            myDlg = New opDeleg(AddressOf mult)
        End If
        Dim a As Single = Convert.ToSingle(TextBox1.Text)
        Dim b As Single = Convert.ToSingle(TextBox2.Text)
```

Aufruf des *Delegate*-Objekts und Ergebnisanzeige:

```
        Dim res As Double = myDlg(a, b)
        TextBox3.Text = res.ToString
    End Sub
```

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 Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim myDlg As opDeleg = Nothing
        If RadioButton1.Checked Then
            myDlg = AddressOf add
        Else
            myDlg = AddressOf mult
        End If
        ...
    End Sub
```

Alle drei bisherigen Varianten erforderten den Aufruf der (benannten) Methoden *add()* und *mult(),* die wir oben implementiert haben. Ganz ohne diese Methoden kommt die folgende Variante aus, die so genannte *Lambda Expression*s verwendet:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim myDlg As opDeleg = Nothing
```

Die Instanziierung der *Delegate*-Objekte erfolgt mittels Lambda-Expressions (diese werden durch das Schlüsselwort *Function* gekennzeichnet):

```
        If RadioButton1.Checked Then
            myDlg = Function(x As Double, y As Double) x + y        ' Lambda-Expr.
        Else
            myDlg = Function(x As Double, y As Double) x * y        ' Lambda-Expr.
        End If
        ...
    End Sub

End Class
```

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 dieser vier Varianten muss zum exakt gleichen Ergebnis führen.

5.8.3 Mit dynamischem Objekt eine Datei durchsuchen

Dieses Beispiel soll das Prinzip der dynamischen Programmierung veranschaulichen (siehe Abschnitt 5.6). Wir wollen mit einem benutzerdefinierten dynamischen Objekt den Inhalt einer Textdatei durchsuchen. Die zu suchende Zeichenkette wird durch den Namen einer *dynamischen Eigenschaft* festgelegt, ruft man z.B. *dynObj.Autor* auf, so wird eine generische Stringliste mit allen Zeilen geliefert, die mit "Autor" beginnen. Die dynamische Klasse unterstützt ein dynamisches Argument, es legt fest, ob vor der Suche führende oder nachfolgende Leerzeichen von jeder Zeile entfernt werden sollen.

Oberfläche

Ein Windows Forms-Formular *Form1* mit einer *ListBox* und einem *Button* genügen.

Quellcode CDynFile

Fügen Sie zum Projekt eine neue Klasse *CDynFile* hinzu.

```
Imports System.IO
```

Für die dynamische Programmierung ist der Namespace *System.Dynamic* einzubinden, über den die abstrakte Klasse *DynamicObject* verfügbar ist.

```
Imports System.Dynamic
```

Unsere Klasse muss von *DynamicObject* erben:

```
Public Class CDynFile
    Inherits DynamicObject
    Private _pfad As String
```

Im Konstruktor wird der Pfad für die zu durchsuchende Textdatei übergeben:

```
    Public Sub New(pfad As String)
        If Not File.Exists(pfad) Then
            Throw New Exception("Datei nicht vorhanden!")
        End If
        _pfad = pfad
    End Sub
```

Die folgende Funktion liefert eine generische *String*-Liste, welche all die Zeilen der Textdatei enthält, in denen die gesuchte Zeichenkette vorkommt. Als Parameter *propName* nimmt die Funktion den Namen der gesuchten Zeichenkette entgegen, dieser tritt hier als dynamische "Eigenschaft" auf[1]. Der Parameter *trimLZ* legt fest, ob die Leerzeichen vor und nach der Zeile, die den Suchtext enthält, entfernt werden sollen.

```
    Public Function getPropValue(propName As String,
                                 Optional trimLZ As Boolean = True) As List(Of String)
```

[1] An eine solche freizügige Definition des Eigenschaftsbegriffs muss man sich erst gewöhnen, dies ist aber typisch für die dynamische Programmierung.

```
        Dim strR As StreamReader = Nothing
        Dim result As New List(Of String)
        Dim line = String.Empty
        Dim lineGB = String.Empty
        Try
            strR = New StreamReader(_pfad)
```

Zeilenweises Einlesen der Textdatei:

```
            While Not strR.EndOfStream
                line = strR.ReadLine
```

Suche in Zeile ohne Beachtung der Groß-/Kleinschreibung:

```
                If trimLZ Then line = line.Trim      ' führende und abschließende LZ entfernen
                lineGB = line.ToUpper                ' in Großbuchstaben verwandeln
                If lineGB.Contains(propName.ToUpper) Then result.Add(line)
            End While
        Catch
            result = Nothing
        Finally
            If strR IsNot Nothing Then strR.Close()
        End Try
        Return result
    End Function
```

Um Runtime-Funktionalität bereitzustellen folgt eine Überschreibung der *TryInvokeMember*-Methode der abstrakten *DynamicObject*-Klasse. Diese Methode wird aufgerufen, wenn ein Mitglied einer dynamischen Klasse angefordert und dabei Argumente übergeben werden:

```
    Public Overrides Function TryInvokeMember(binder As InvokeMemberBinder,
                                args() As Object, ByRef result As Object) As Boolean
```

Zur Bedeutung der Parameter:

binder: Infos über das referenzierte Mitglied, z.B. dessen Name

result: Resultat des spezifizierten Mitglieds (per Referenz!)

args: Array von Argumenten, die beim Aufruf der dyn. Eigenschaft übergeben werden können

```
        Dim trim = True
        Try
            If args.Length > 0 Then trim = Convert.ToBoolean(args(0))
        Catch
            MessageBox.Show("Trim-Argument muss Boole'scher Wert sein!")
        End Try
```

Hier erfolgt der Aufruf obiger *getPropValue*-Funktion mit Übergabe des Namens der gesuchten Zeichenkette (= Eigenschaft!) und ob LZ-Entfernung gewünscht:

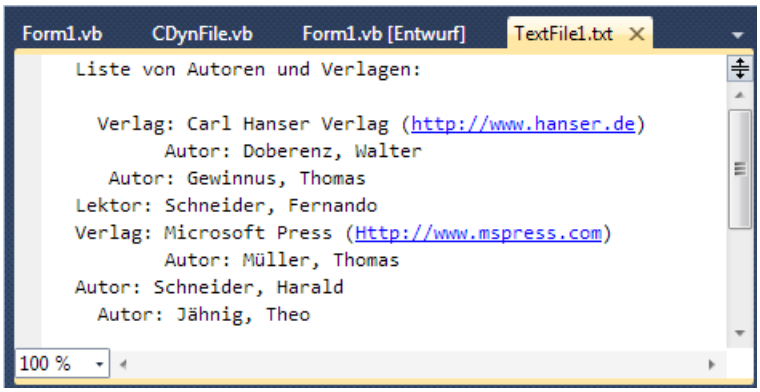
```
        result = getPropValue(binder.Name, trim)
```

True wird zurückgegeben, wenn das referenzierte Mitglied vorhanden ist, ansonsten *False*:

```
        Return If(result Is Nothing, False, True)
    End Function

End Class
```

Textdatei hinzufügen

Klicken Sie auf *Projekt/Neues Element hinzufügen...* und wählen Sie im Dialogfenster "Textdatei", die Sie auf dem standardmäßigen Namen "TextFile1" belassen können. Tragen Sie in die Textdatei mehrere Zeilen in der Art einer Liste ein (siehe folgende Abbildung).



Quellcode Form1

```
Option Strict Off                    ' wegen Late Binding!

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Das dynamische Objekt wird erzeugt:

```
        Dim dynObj As Object = New CDynFile("..\..\TextFile1.txt")
```

Das dynamische Objekt wird benutzt:

Suche nach "Doberenz" überall in der Zeile (Standardsuche, führende und nachfolgende LZ werden entfernt):

```
        For Each line In dynObj.Doberenz
            ListBox1.Items.Add(line) '
        Next

        ListBox1.Items.Add("-----------------------")
```

Suche nach "Autor" überall in der Zeile, führende und nachfolgende LZ werden **nicht** entfernt:

```
        For Each line In dynObj.Autor(False)
            ListBox1.Items.Add(line)
        Next
        ...
```

Hier können Sie weitere "Suchexperimente" einfügen:

```
        ...
    End Sub
End Class
```

Test

So etwa dürfte das Ergebnis aussehen[1]:

```
Autor: Doberenz, Walter

Liste von Autoren und Verlagen:
      Autor: Doberenz, Walter
   Autor: Gewinnus, Thomas
      Autor: Müller, Thomas
Autor: Schneider, Harald
  Autor: Jähnig, Theo

Lektor: Schneider, Fernando
```

Bemerkungen

- Dynamische Objekte verlangen spätes Binden und das geht leider nicht mit eingeschalteter strenger Typprüfung, deshalb ist (zumindest für die Klasse *Form1*) die *Option Strict Off* einzuschalten.
- Der Nachteil des späten Bindens ist, dass Sie bei der Auswahl eines Mitglieds für ein dynamisches Objekt keinerlei IntelliSense-Unterstützung haben, also "auf gut Glück" zuweisen müssen.

[1] Wie Sie sehen, haben wir hier auch unserem langjährigen Lektor ein Denkmal gesetzt.

Kapitel 6

Einführung in LINQ

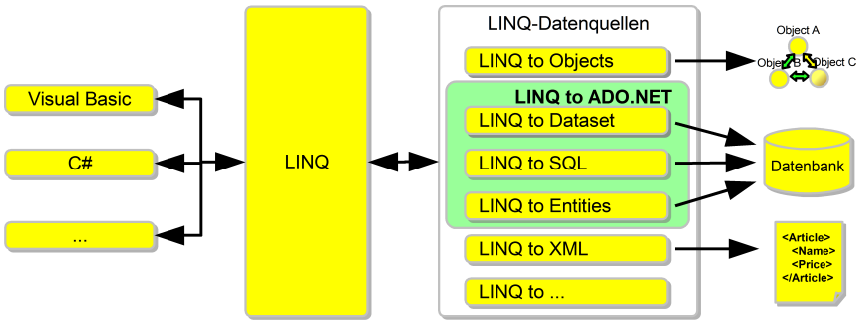
LINQ (Language Integrated Query) ist ein relativ junges Visual Basic-Sprachfeature. Bei *LINQ to Objects* handelt es sich um die allgemeinste und grundlegendste LINQ-Implementierung, welche auch die wichtigsten Bausteine für die übrigen LINQ-Implementierungen liefert. In einer SQL-ähnlichen Syntax können miteinander verknüpfte Collections/Auflistungen abgefragt werden, die über die *IEnumerable*-Schnittstelle verfügen.

6.1 LINQ-Grundlagen

Der wichtigste Vorteil von LINQ ist die standardisierte Möglichkeit, nicht nur Tabellen in einer relationalen Datenbank abzufragen, sondern auch Textdateien, XML-Dateien und andere Datenquellen, die eine identische Syntax verwenden. Ein zweiter Vorteil ist die Fähigkeit, diese standardisierten Methoden von jeder .NET-konformen Sprache (wie zum Beispiel VB oder C#) 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 (momentan für C# und VB).

6.1.2 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 werden als Bestandteil 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 dieser Provider die grundlegende Technologie bereitstellt und die übrigen Flavours mehr eine Angelegenheit der Datenbankliteratur sind.

Nochmals sei hier betont, dass LINQ eine offene Technologie ist, der jederzeit neue Provider hinzugefügt werden können. Die im .NET-Framework enthaltenen Implementierungen 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 kurz eingehen.

6.1.3 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 eines anonymen Objekt-Initialisierers, 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.

VB
```
Dim person = New With {.Nachname = "Müller", .Alter = 53}
```

Der Compiler generiert hierfür intern (in MSIL-Code) die folgende Klasse:

```
Friend Class ??????
    Private _nachname As String
    Private _alter As Integer

    Public Property Nachname() As String
        Get
            Return _nachname
        End Get
        Set(value As String)
            _nachname = value
        End Set
    End Property

    Public Property Alter() As Integer
        Get
            Return _alter
        End Get
        Set(value As Integer)
            _alter = value
        End Set
    End Property
End Class
```

HINWEIS: Da der Typ der Eigenschaften aus der jeweiligen Klasse bzw. Struktur des im Objekt-Initialisierer zugewiesenen Wertes abgeleitet wird, darf man hier nicht *Nothing* 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.

VB
```
Dim person = New With {.Nachname = "Müller", .Alter = Nothing}      ' 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.

VB
```
Dim kunde = New With {.Nachname = "Krause", .Alter = 29}
person = kunde
```

6.1.4 Erweiterungsmethoden

Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Ab VB 9.0 wurde eine neue Syntax eingeführt, die das direkte Hinzufügen neuer Methoden zu einer vorhandenen Klasse erlaubt. Mit anderen Worten: Mit Erweiterungsmethoden können Sie einem Datentyp oder einer Schnittstelle Methoden außerhalb der Definition hinzufügen.

HINWEIS: In Visual Basic müssen sowohl die Erweiterungsmethode als auch das Modul, welches die Erweiterungsmethode enthält, mit dem Attribut *System.Runtime.CompilerServices.Extension* versehen werden.

BEISPIEL 6.4: Die Klasse *System.Int32* wird um die Methoden *mult()* und *abs()* erweitert.

VB
```
Imports System.Runtime.CompilerServices
...
<Extension()>
Public Module IntExtension
    <Extension()>
    Public Function mult(i As Integer, faktor As Integer) As Integer
        Return i * faktor
    End Function

    <Extension()>
    Public Function abs(i As Integer) As Integer
        If i < 0 Then i = -1 * i
        Return i
    End Function
End Module
```

Der Test:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim zahl As Integer = -95
        TextBox1.Text = zahl.mult(7).ToString             ' -665
        TextBox2.Text = zahl.abs.ToString                 '   95
    End Sub
```

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 Instanzen-Methoden 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.

Erweiterungsmethoden ermöglichen es Ihnen, einem vorhandenen Typ neue Methoden hinzuzufügen, ohne den Typ tatsächlich zu ändern. Die Standardabfrageoperatoren in LINQ stellen eine Reihe von Erweiterungsmethoden dar, die Abfragefunktionen für jeden Typ bieten, der die generische *IEnumerable(Of T)*-Schnittstelle implementiert.

BEISPIEL 6.5: Durch die folgende Erweiterungsmethode wird der *String*-Klasse eine *Print*-Methode hinzugefügt.

VB
```
<Extension()>
Public Sub Print(s As String)
    Console.WriteLine(s)
End Sub
```

Die Methode wird wie jede normale Instanzenmethode von *String* aufgerufen:

```
Dim msg As String = "Hallo"
msg.Print()
```

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 (*Order By*). 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 6.1.4) definiert und in der Regel auf beliebige Objekte anwendbar, die das generische Interface *IEnumerable(Of T)* implementieren.

BEISPIEL 6.6: Gegeben sei die Auflistung

VB
```
Dim 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.

```
Dim expr = From s In monate
                  Where s.Length = 6
                  Order By s
```

BEISPIEL 6.6: Gegeben sei die Auflistung

```
                    Select s.ToUpper
```

Die Ergebnisanzeige:

```
For Each item As String In expr
    ListBox1.Items.Add(item)
Next item
```

Ergebnis

Das Resultat:

```
AUGUST
JANUAR
```

Obiges Beispiel demonstriert das allgemeine Format einer LINQ-Abfrage:

SYNTAX: `From ... < Where ... Order By ... > 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

```
Dim expr = From s In monate ...
```

ist für den Compiler identisch mit

```
Dim expr As IEnumerable(Of String) = 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 (Abfrage-Syntax)
- Extension Method-Syntax[1] (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.

[1] Die *Extension Method-Syntax* wird auch als *Dot Notation Syntax* bezeichnet.

BEISPIEL 6.7: Die LINQ-Abfrage des obigen Beispiels in Extension Method-Syntax.

VB
```
Dim expr = monate.Where(Function(s) s.Length = 6).
           OrderBy(Function(s) s).
           Select(Function(s) s.ToUpper())
```

Oder kompakt in einer Zeile:

```
Dim expr = monate.Where(Function(s) s.Length = 6).OrderBy(Function(s) s).
                                                                Select(Function(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 6.8: Obiges Beispiel in gemischter Syntax:

VB
```
Dim expr = (From s In monate
           Where s.Length = 6
           Select s.ToUpper()).
           OrderBy(Function(s) s)
```

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*, *Order By*, *Group By* 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.

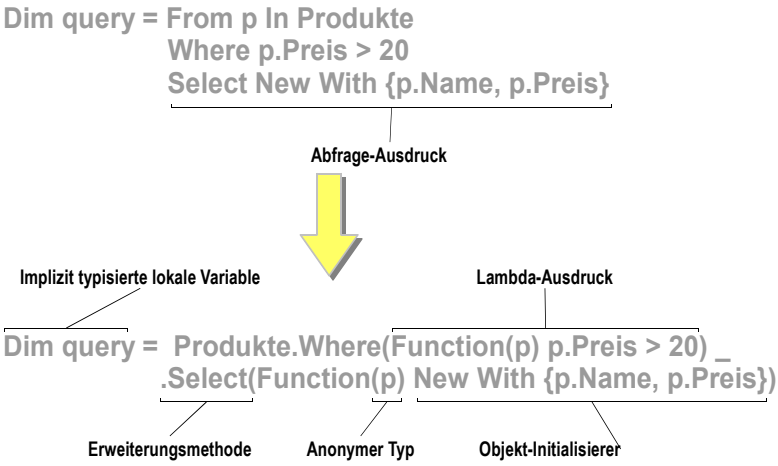
6.2.3 Übersicht der wichtigsten Abfrage-Operatoren

Die Klasse *Enumerable* im Namespace *System.Linq* stellt zahlreiche Abfrageoperatoren für LINQ to Objects bereit und definiert diese als Erweiterungsmethoden für Typen die *IEnumerable(Of 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, Select Many*
Sortieroperatoren (Ordering)	*Order By, Then By*
Gruppierungsoperatoren (Grouping)	*GroupBy*
Quantifizierungsoperatoren (Quantifiers)	*Any, All, Contains*
Aufteilungsoperatoren (Partitioning)	*Take, Skip, Take While, Skip While*
Mengenoperatoren (Sets)	*Distinct, Union, Intersect, Except*
Elementoperatoren (Elements)	*First, FirstOrDefault, ElementAt*
Aggregatoperatoren (Aggregation)	*Count, Sum, Min, Max, Average*
Konvertierungsoperatoren (Conversion)	*ToArray, ToList, ToDictionary*
Typumwandlungsoperatoren (Casting)	*OfType T*



Die obige 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.

Der Let-Zuweisungsoperator

Um innerhalb einer LINQ-Abfrage bestimmte Zwischenergebnisse zuweisen zu können, verwenden Sie den *Let*-Operator.

BEISPIEL 6.9: LINQ-Abfrage zum Durchsuchen eines Verzeichnisbaums

VB
```
Dim qryMatchFiles = From f In fList
                    Where f.Extension = ext
                    Let fText = File.ReadAllText(f.FullName)
                    Let matches = rgx.Matches(fText)
                    Where (matches.Count > 0)
                    Select Name = f.FullName,
                           Matches = From match As Match In matches
                               Select match.Value
...
```

Das komplette Programm finden Sie im Praxisbeispiel

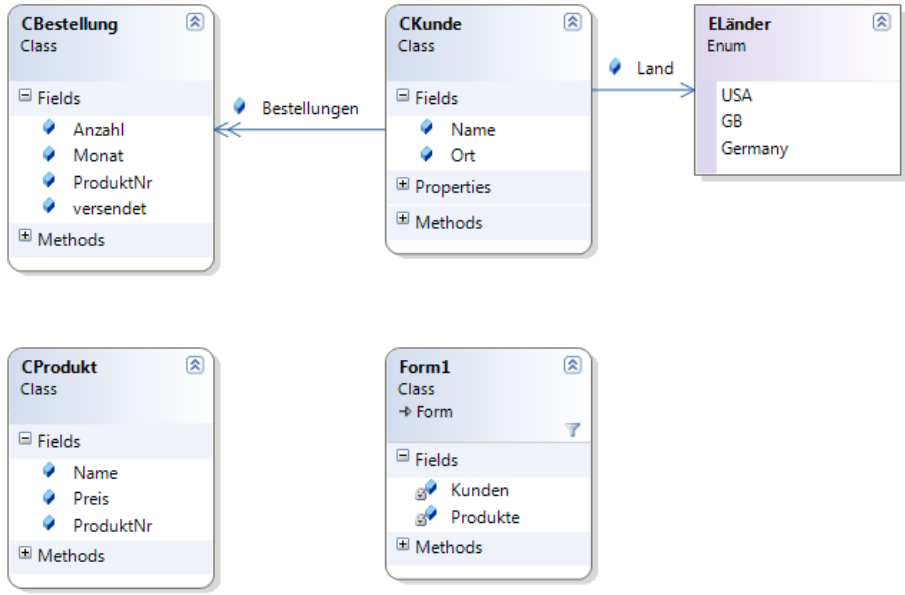
► 7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen

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(Of T)* implementiert.

BEISPIEL 6.10: Die komplette Produktliste wird ausgegeben.

VB

Zunächst in Extension Method Syntax:

```
Dim allProdukte = Produkte.Select(Function(p) p.Name)
```

Alternativ die Query Expression-Syntax:

```
Dim allProdukte = From p In Produkte Select p.Name
```

Die Ausgabe der Ergebnisliste:

```
For Each p In allProdukte
        ListBox1.Items.Add(p)
Next
```

BEISPIEL 6.10: Die komplette Produktliste wird ausgegeben.

Ergebnis

Der Inhalt der *ListBox* sollte dann etwa folgenden Anblick bieten:

```
Marmelade
Quark
Mohrrüben
...
```

BEISPIEL 6.11: Das Abfrageergebnis wird auf einen anonymen Typ projiziert, der als Tupel (Datensatz) definiert ist.

VB

```
Dim expr = Kunden.Select(Function(k) New With {k.Name, k.Ort})
```

Die Ausgabeschleife:

```
For Each k In expr
    ListBox1.Items.Add(k)
Next
```

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(Of 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 6.12: Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

VB

```
Dim bestellungen = Kunden.Where(Function(k) k.Land = ELänder.Germany).
                                        SelectMany(Function(k) k.Bestellungen)
```

Alternativ der Abfrageausdruck in Query Expression Syntax:

```
Dim bestellungen = From k In Kunden
                   Where k.Land = ELänder.Germany
                   From b In k.Bestellungen
                   Select b
```

Das Auslesen des Ergebnisses der Abfrage:

```
For Each b In bestellungen
    ListBox1.Items.Add(b)
Next
```

BEISPIEL 6.12: Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

Ergebnis

Das Ergebnis (Voraussetzung ist eine 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 6.13: Die Kunden an den Positionen 2 und 3 der Kundenliste sollen angezeigt werden.

VB

```
Dim expr = Kunden.Where(Function(k, index) (index >= 2) And (index < 4)).
                                                    Select(Function(k) k.Name)
```

Die Ausgabe:

```
For Each kd In expr
    ListBox1.Items.Add(kd)
Next
```

Ergebnis

Das Ergebnis:

```
Holger
Fernando
```

6.3.3 Die Sortierungsoperatoren OrderBy und ThenBy

Diese Operatoren bewirken ein Sortieren der Elemente innerhalb der Ergebnismenge.

OrderBy/OrderByDescending

Das Pärchen ermöglicht Sortieren in auf- bzw. absteigender Reihenfolge.

BEISPIEL 6.14: Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (der teuerste zuerst).

VB

```
Dim prod = Produkte.
           Where(Function(p) p.Preis <= 20).
           OrderByDescending(Function(p) p.Preis).
           Select(Function(p) New With {p.Name, p.Preis})
```

Oder alternativ als Abfrageausdruck:

```
Dim prod = From p In Produkte
           Where (p.Preis <= 20)
```

BEISPIEL 6.14: Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (der teuerste zuerst).

```
            Order By p.Preis Descending
            Select p.Name, p.Preis
```

Die Ausgabeschleife:

```
For Each p In prod
    ListBox1.Items.Add(p)
Next
```

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(Of T)*, sondern nur auf den Typ *IOrderedSequence(Of T)* anwendbar sind, können diese Operatoren nur im Anschluss an *OrderBy*/ *OrderByDescending* eingesetzt werden.

BEISPIEL 6.15: Alle Kunden sollen zunächst nach ihrem Land und dann nach ihren Namen sortiert werden.

VB

```
Dim knd = Kunden.OrderBy(Function(k) k.Land).
          ThenBy(Function(k) k.Name).
          Select(Function(k) New With {k.Land, k.Name})
```

Der alternative Abfrageausdruck:

```
Dim knd = From k In Kunden
          Order By k.Land, k.Name
          Select k.Land, k.Name
```

Die Ausgabe

```
For Each ku In knd
    ListBox1.Items.Add(ku)
Next
```

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 6.16: Das Vorgängerbeispiel mit umgekehrter Reihenfolge der Ergebniselemente:

VB
```
Dim knd = Kunden.OrderBy(Function(k) k.Land).
          ThenBy(Function(k) k.Name).
          Select(Function(k) New With {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 6.17: Alle Kunden nach Ländern gruppieren

VB
```
Dim knd = Kunden.GroupBy(Function(k) k.Land)
```

Der alternative Abfrageausdruck:

```
Dim knd = From k In Kunden Group By k.Land Into Group
```

Durchlaufen der Ergebnismenge:

```
For Each kdGroup In knd
    ListBox1.Items.Add(kdGroup.Land)
    For Each kd In kdGroup.Group
        ListBox1.Items.Add("  " & kd.Name)
    Next
Next
```

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(Of IGrouping(Of K, T)* liefern. Die generische Schnittstelle *IGrouping(Of K, T)* definiert einen spezifischen Schlüssel vom Typ *K* für die Gruppenelemente (Typ *T*).

BEISPIEL 6.18: Alle Produkte werden nach ihren Anfangsbuchstaben gruppiert

VB

```
Dim prodGroups = Produkte.GroupBy(Function(p) p.Name(0), Function(p) p.Name)
```

Die (verschachtelte) Ausgabeschleife:

```
For Each pGroup In prodGroups
    ListBox1.Items.Add(pGroup.Key)
    For Each p In pGroup
        ListBox1.Items.Add("  " & p)
    Next
Next
```

Ergebnis

```
M
  Marmelade
  Mohrrüben
  Mehl
Q
  Quark
K
  Käse
H
  Honig
```

BEISPIEL 6.19: Zum gleichen Resultat wie das Vorgängerbeispiel führt der folgende Code unter Verwendung eines Abfrageausdrucks

VB

```
Dim prodGroups = From p In Produkte
                 Group By FirstLetter = p.Name(0)
                 Into prods = Group
```

Die (verschachtelte) Ausgabeschleife:

```
For Each pGroup In prodGroups
    ListBox1.Items.Add(pGroup.FirstLetter)
    For Each pr In pGroup.prods
        ListBox1.Items.Add("  " & pr.Name)
    Next
Next
```

6.3.5 Verknüpfen mit Join

Mit diesem Operator definieren Sie Beziehungen zwischen verschiedenen Auflistungen. Im folgenden Beispiel werden Bestelldaten auf Produkte projiziert.

BEISPIEL 6.20: Die Bestellungen aller Kunden werden aufgelistet.

VB

```
Dim bestprod = Kunden.
               SelectMany(Function(k) k.Bestellungen).
               Join(Produkte, Function(b) b.ProduktNr,
               Function(p) p.ProduktNr, Function(b, p) New With {b.Monat, p.ProduktNr,
               p.Name, p.Preis, b.versendet})
```

Alternativ die Notation in Abfragesyntax:

```
Dim bestprod = From k In Kunden
               From b In k.Bestellungen
               Join p In Produkte On b.ProduktNr Equals p.ProduktNr
               Select New With {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:

```
For Each bp In bestprod
    ListBox1.Items.Add(bp)
Next
```

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 6.21: Alle Kunden sollen, zusammen mit der Anzahl der von ihnen aufgegebenen Bestellungen, angezeigt werden.

VB

```
Dim kdn = Kunden.
            Select(Function(k) New With {k.Name, k.Ort, .AnzahlBest = k.Bestellungen.Count})
```

BEISPIEL 6.21: Alle Kunden sollen, zusammen mit der Anzahl der von ihnen aufgegebenen Bestellungen, angezeigt werden.

Oder das Gleiche in Abfrage-Syntax:

```
Dim kdn = From k In Kunden
          Select k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count()
```

Wir iterieren durch die Ergebnismenge:

```
For Each ku In kdn
    ListBox1.Items.Add(ku)
Next
```

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 6.22: Die Summe aller Preise der Produktliste

VB

```
Dim total = Produkte.Sum(Function(p) p.Preis)
```

Die alternative Abfrage-Syntax (eigentlich gemischte Syntax):

```
Dim 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 6.23: Die Gesamtsumme der von allen Kunden aufgegebenen Bestellungen wird ermittelt

VB

```
Dim 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 With {k.Name, .BestellBetrag = b.Anzahl * p.Preis})
            On k.Name Equals b.Name
```

BEISPIEL 6.23: Die Gesamtsumme der von allen Kunden aufgegebenen Bestellungen wird ermittelt

```
            For Each k In expr
                ListBox1.Items.Add(k.b)
            Next
```

Ergebnis

```
{Name = Walter, BestellBetrag = 40}
{Name = Thomas, BestellBetrag = 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 6.24: Alle Produkte, die mit dem Buchstaben "M" beginnen, sollen ermittelt werden.

VB

```
Dim MProds = From p In Produkte Where p.Name(0) = "M" Select p.Name
```

Die Ergebnismenge wird das erste Mal durchlaufen und angezeigt:

```
For Each prod In MProds
    ListBox1.Items.Add(prod)
Next 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:

```
For Each prod In MProds
    ListBox1.Items.Add(prod)
Next 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 neu ausgeführt wird, wenn wir (wie hier in der *For Each*-Schleife) auf das Abfrageergebnis (*MProds*) zugreifen.

Abfragen dieser Art bezeichnet man deshalb auch als "verzögerte Abfragen". 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 *For Each*-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 6.25: Das Vorgängerbeispiel wird wiederholt, diesmal aber wird das Abfrageergebnis in einem Array zwischengespeichert.

VB

```
Dim MProds = (From p In Produkte Where p.Name(0) = "M" Select p.Name).ToArray
...
```

Die Änderung der Quellfolge bleibt jetzt ohne Konsequenz für das Abfrageergebnis:

```
Produkte(0).Name = "Milch"
...
```

Ergebnis

```
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.

HINWEIS: In vielen Szenarien kann PLINQ signifikant die Geschwindigkeit von LINQ to Objects-Abfragen steigern, da es alle verfügbaren Prozessoren des Computers effizient 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(Of T)* implementiert.

Der Aufruf von *AsParallel* veranlasst den VB-Compiler, die parallele Version der Standard-Abfrageoperatoren zu binden. Damit übernimmt PLINQ die weitere Verarbeitung der Abfrage.

BEISPIEL 6.26: Eine einfache LINQ-Abfrage über eine Liste von Integer-Zahlen

VB

```
Dim zahlen = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Oder auch:

```
Dim q = From x in zahlen
        Where x > 3
        Order By x Descending
        Select x
```

Erst beim Iterieren über die Liste wird die Abfrage ausgeführt:

```
For Each z In q
    ListBox1.Items.Add(z.ToString)            ' 9, 8, 7, 6, 5, 4
Next
```

Um dieselbe Abfrage mittels PLINQ auszuführen, ist lediglich *AsParallel* auf den Daten aufzurufen:

BEISPIEL 6.27: Die Abfrage im obigen Beispiel mit PLINQ

VB

```
...
Dim q = From x in zahlen.AsParallel
        Where x > 3
        Order By 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 6.28: Beide obigen Abfragen in Erweiterungsmethoden-Syntax

VB

Einfache LINQ-Version:

```
...
Dim q = zahlen.
        Where(Function(x) x > 3).
        OrderByDescending(Function(x) x).
        Select(Function(x) x)
...
```

PLINQ-Version:

```
...
Dim q = zahlen.AsParallel.
        Where(Function(x) x > 3).
        OrderByDescending(Function(x) x).
        Select(Function(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 *For Each*-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 ohnehin 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 6.29: Verwendung von *AsOrdered*

VB

```
Dim zahlen = {7, 4, 2, 3, 1, 6, 11, 5, 10, 8, 9, 13, 12}
Dim q = (From x In zahlen.AsParallel.AsOrdered
         Where x > 3
         Select x).
         Take(5)                              ' nur die ersten fünf
```

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 könnte 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/en-us/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 Beispiel lernen Sie den prinzipiellen Aufbau von LINQ[1]-Abfragen kennen. Im Zusammenhang damit kommen 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 zur Anwendung.

[1] *Language Integrated Query*

Im Folgenden werden beide Syntaxformen demonstriert, 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

In der Regel ist der *System.Linq* Namespace bereits standardmäßig eingebunden, sodass die folgende Anweisung entfallen kann:

```
Imports System.Linq

Public Class Form1
```

Das abzufragende Integer-Array enthält irgendwelche ganzzahligen Werte:

```
    Private zahlen() As Integer = {5, -4, 18, 26, 0, 19, 16, 2, -1, 0, 9, -5, 8, 15, 19}
```

Die Abfrage in Query Expression Syntax:

```
     Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

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*, *Order By*, *Select*) zum Einsatz:

```
        Dim expr = From z In zahlen
                   Where z > 10
                   Order By z
                   Select z
```

HINWEIS: Obige Definition hat noch nicht die Ausführung der Abfrage zur Folge (siehe Bemerkungen)! Gleiches gilt auch für die späteren zwei Abfrageausdrücke.

Ausführen der Abfrage und Anzeige des Abfrageergebnisses:

```
        For Each z As Integer In expr
            ListBox1.Items.Add(z.ToString)
        Next z
    End Sub
```

Dieselbe Abfrage in Extension Method Syntax:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Hier werden im Abfrageausdruck so genannte Erweiterungsmethoden (*Where*, *OrderBy*, *Select*) zusammen mit Lambda-Ausdrücken (*Function(z)* z) benutzt:

```
    Dim expr = zahlen.Where(Function(z) z > 10).
               Order By(Function(z) z).
```

```
            Select(Function(z) z)
```

Ausführen der Abfrage und Anzeige des Abfrageergebnisses:

```
        For Each z As Integer In expr
            ListBox1.Items.Add(z.ToString)
        Next z
    End Sub
```

Schließlich die gleiche Abfrage in gemischter Syntax:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
```

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:

```
        Dim expr = (From z In zahlen
                    Where z > 10
                    Select z).
                    OrderBy(Function(z) z)
```

Ausführen der Abfrage und Anzeige des Abfrageergebnisses:

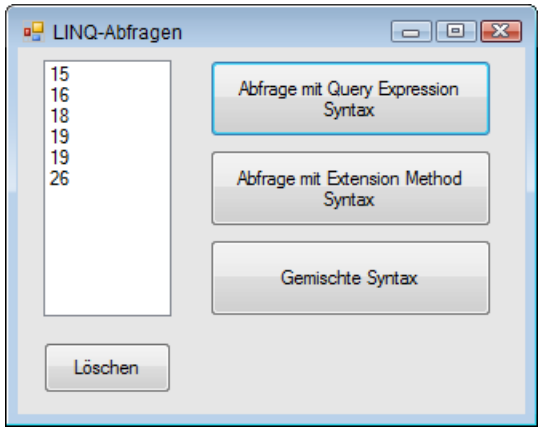
```
        For Each z As Integer In expr
            ListBox1.Items.Add(z.ToString)
        Next z
    End Sub
```

ListBox-Inhalt löschen:

```
     Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        ListBox1.Items.Clear()
    End Sub
End Class
```

Test

Egal, welche der drei Schaltflächen Sie klicken, das Ergebnis wird stets dasselbe sein:



Bemerkungen

- Die Abfrage wird mit Definition des Abfrageausdrucks *expr* noch nicht ausgeführt! Sie können sich leicht selbst davon überzeugen, indem Sie **nach** Definition von *expr* den Inhalt der *zahlen*-Collection manipulieren. Das angezeigte Ergebnis bezieht sich immer auf den aktuellen Inhalt der Collection.
- 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 Wie wir gesehen haben, erweist sich dieses Feature vor allem für *anonyme Typen* als praktisch bzw. notwendig.
- Bei den *Lambda-Ausdrücke*n handelt es sich vom Prinzip her um nichts weiter als um funktional erweiterte *anonyme Methoden*. Der offensichtlichste Unterschied zeigt sich in der Syntax, auf die Parameterliste folgt das Schlüsselwort *Function*, danach kommt eine einzelne Anweisung oder ein Anweisungsblock.
- Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. VB führt eine 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. Instanzmethode) des erweiterten Datentyps aufgerufen werden.
- Nicht nur Zahlen-Auflistungen, wie im vorliegenden Beispiel, sondern jede Collection, die das *System.Collections.Generic.IEnumerable* Interface (oder das generische Interface *IEnumerable(Of T)* unterstützt, kann mit LINQ verarbeitet werden.

6.4.2 Aggregat-Abfragen mit LINQ

Im vorliegenden Beispiel 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 (siehe Laufzeitansicht).

Quellcode

```
Public Class Form1

Start:

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Länge der Zahlenliste wird bestimmt:

```
        Dim n As Integer = ListBox1.Items.Count       ' Anzahl der Werte
        If n = 0 Then Exit Sub
```

Zahlen werden aus der *ListBox* an ein *Single*-Array übergeben:

```
        Dim zahlen(n - 1) As Single
        For i As Integer = 0 To n - 1
            zahlen(i) = Convert.ToSingle(ListBox1.Items(i))
        Next i
```

Die Abfrage wird definiert:

```
        Dim expr = From z In zahlen
                   Order By z
                   Select z
```

Nacheinander wird der Abfrage-Ausdruck für verschiedene Aggregat-Methoden ausgewertet:

```
        Dim count = expr.Count()              ' Anzahl
        Label1.Text = count.ToString
        Dim sum = expr.Sum()                  ' Summe
        Label2.Text = sum.ToString
        Dim avg = expr.Average()              ' Durchschnitt
        Label3.Text = avg.ToString
        Dim max = expr.Max()                  ' Maximum
        Label4.Text = max.ToString
        Dim min = expr.Min()                  ' Minimum
        Label5.Text = min.ToString
```

Sortierte Anzeige in der *ListBox*:

```
        ListBox1.Items.Clear()
        For Each z As Single In expr
            ListBox1.Items.Add(z.ToString)
        Next z
    End Sub
```

Eine in die *TextBox* eingegebene Zahl wird mittels Enter-Taste in die *ListBox* übernommen:

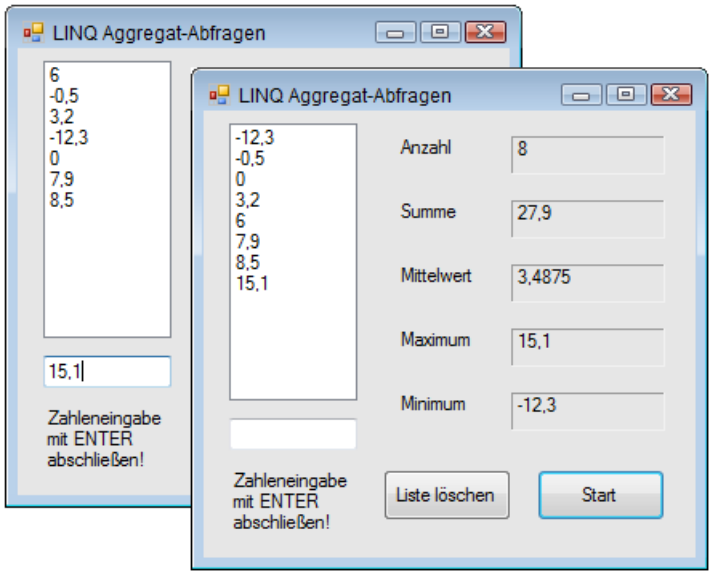
```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter Then
            ListBox1.Items.Add(TextBox1.Text)
            TextBox1.Clear()
        End If
    End Sub
```

ListBox-Inhalt löschen:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        ListBox1.Items.Clear()
    End Sub
End Class
```

Test

Nach Eingabe einer Zahlenkolonne (Dezimaltrennzeichen ist natürlich das Komma!) können Sie Auswertung und Sortierung starten:



Bemerkungen

- Um die Zahlenreihe anstatt in aufsteigender, in absteigender Reihenfolge zu sortieren, ist im Abfrageausdruck die Klausel ***Order By z*** zu ersetzen durch ***Order By z Descending***.
- Die Wirkung der Aggregatmethoden (*Count, Sum, Average, Min, Max*) ist völlig unabhängig davon, ob die zugrunde liegende Zahlenmenge sortiert ist oder nicht.

6.4.3 LINQ im Schnelldurchgang erlernen

Ganz ernst zu nehmen ist der Titel natürlich nicht, denn LINQ ist eine komplexe Technologie und der Einsteiger kann sie mittels eines solch kurzen Beispiels bestenfalls "beschnuppern".

Worum also geht es bei LINQ?

Mit LINQ lassen sich Elemente aus einer Auflistung oder einem Array selektieren, wobei eine SQL-ähnliche Syntax benutzt wird. Für die Erstellung einer einfachen LINQ-Abfrage sind folgende grundlegende Schritte auszuführen:

1. Beginnen Sie eine neue LINQ-Abfrage mit dem Schlüsselwort *From*.
2. Identifizieren Sie die Filterbedingungen mit dem Schlüsselwort *Where*.

3. Legen Sie mit dem Schlüsselwort *Select* fest, welche Werte der gefilterten Menge im Ergebnis der Abfrage enthalten sein sollen.
4. Bestimmen Sie die Sortierreihenfolge mit dem Schlüsselwort *Order By*.

Das Ergebnis einer LINQ-Abfrage ist eine Instanz von *IEnumerable* und enthält die Collection- bzw. Arrayelemente, die den Filterbedingungen entsprechen. Sie können *IEnumerable* verwenden, um mittels *foreach*-Schleife über die Abfrageergebnisse zu iterieren, oder auch es als Datenquelle für weitere LINQ-Abfragen einsetzen.

Oberfläche

Ein Formular mit vier Schaltflächen und einer *ListBox*.

Quellcode und Tests

Die Abfrage wird einmal mit Schlüselwörtern (*Query Expression Syntax*) und ein zweites Mal mit Erweiterungsmethoden und Lambda-Ausdrücken (*Extension Method Syntax*) ausgeführt.

```
Public Class Form1
```

Das *String*-Array, welches wir mit LINQ abfragen wollen:

```
    Private strArray() As String = {"Müller", "Meier", "Kurze", "Schneider", "Merkel",
                                    "Krause", "Baumann", "Gewinnus", "Doberenz"}
```

Eine einfache LINQ Abfrage:

Wir selektieren die Einträge, die länger als 5 Zeichen sind und sortieren das Ergebnis der Länge nach:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Auf lokaler Ebene können wir von der so genannten "Typinferenz" Gebrauch machen und den Ergebnistyp der Abfrage mit dem Schlüsselwort *Dim* anstatt mit *IEnumerable(Of String)* deklarieren:

```
        Dim linqRes = From ps In strArray
                      Where ps.Length > 5
                      Order By ps.Length
                      Select ps
        anzeigen(linqRes)
    End Sub
```

Dasselbe Ergebnis produziert die folgende Abfrage auf der Basis von Erweiterungsmethoden und Lambda Ausdrücken:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim linqRes = strArray.Where(Function(ps) ps.Length > 5).
                               OrderBy(Function(ps) ps.Length).Select(Function(ps) ps)
        anzeigen(linqRes)
    End Sub
```

Für sehr große Collections und Arrays bietet sich *Parallel LINQ* (PLINQ) an, welches Ihre Abfrage zwischen mehreren Threads aufteilt um sie parallel zu verarbeiten. PLINQ aktivieren Sie, indem Sie die *AsParallel*-Method auf das Array anwenden.

Dieselbe Abfrage mit PLINQ:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    Dim linqRes = From ps In strArray.AsParallel
                  Where ps.Length > 5
                  Order By ps.Length
                  Select ps
    anzeigen(linqRes)
End Sub
```

Die folgende Hilfsmethode erwartet als Übergabeparameter ein *IEnumerable<String>*, sie iteriert über die Ergebnismenge und erledigt die Anzeige in der *ListBox:*

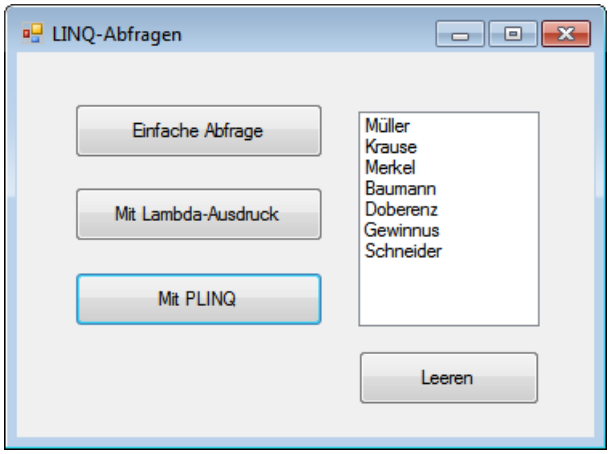
```
Private Sub anzeigen(linqRes As IEnumerable(Of String))
    For Each s As String In linqRes
        ListBox1.Items.Add(s)
    Next
End Sub
```

Eher nebensächlich – das Leeren der *ListBox*:

```
Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
    ListBox1.Items.Clear()
End Sub
End Class
```

Test

Alle drei Abfragen führen zum gleichen Ergebnis:



6.4.4 Strings mit LINQ abfragen und filtern

Auch die *String*-Klasse implementiert die generische *IEnumerable<T>*-Schnittstelle, deshalb kann mittels LINQ jede Zeichenfolge als Zeichensequenz aufgefasst und abgefragt werden. Im vorliegenden Beispiel ermitteln wir auf diese Weise unter anderem die Anzahl der in einem String enthaltenen numerischen Ziffern.

Oberfläche

Auf das Startformular *Form1* setzen Sie eine *ListBox* und einen *Button*.

Quellcode

```
Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Ein x-beliebiger String:

```
        Dim s As String = "XY34/w = abc-12"
        ListBox1.Items.Add(s)
        ListBox1.Items.Add(String.Empty)
```

Mittels LINQ-Abfrage nur die Zeichen auswählen, die Zahlen darstellen:

```
        Dim query1 = From c In s
                     Where Char.IsDigit(c)
                     Select c
```

Erst jetzt wird die Abfrage ausgeführt:

```
        For Each c As Char In query1
            ListBox1.Items.Add(c)
        Next c
```

Die *Count*-Methode auf der Abfrage ausführen:

```
        Dim count As Integer = query1.Count()
        ListBox1.Items.Add("Anzahl = " & count.ToString())
```

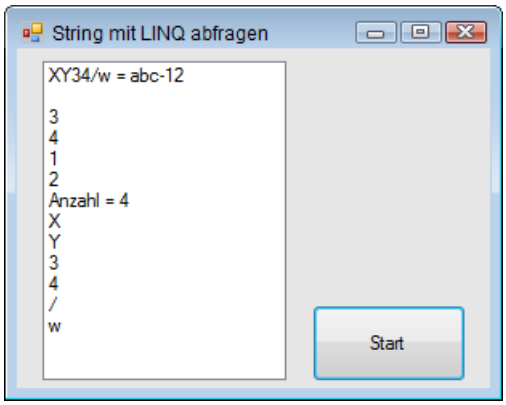
Alle Zeichen vor dem ersten "=" auswählen:

```
        Dim query2 = s.TakeWhile(Function(c) c <> "=")
```

Die zweite Abfrage ausführen:

```
        For Each c As Char In query2
            ListBox1.Items.Add(c.ToString())
        Next c
    End Sub
End Class
```

Test



Bemerkungen

- In unserem Beispiel wird die Abfrage *query1* ein zweites Mal verwendet, nachdem sie erstmalig ausgeführt wurde. Dies ist möglich, da die Abfrage selbst die eigentlichen Ergebnisse nicht speichert.
- Das Beispiel soll vor allem das LINQ-Prinzip verdeutlichen, denn für die praktische Ausführung von Mustervergleichsoperationen wird man besser die *Regex*-Klasse verwenden.

6.4.5 Duplikate aus einer Liste oder einem Array entfernen

Im folgenden Beispiel werden wir aus einer generischen Liste *List(Of CPerson)* mehrfach vorkommende Einträge löschen. Wir verwenden dazu eine benutzerdefinierte Implementierung des *IEqualityComparer*-Interface, welche der *List.Distinct*-Methode übergeben wird.

Die Realisierung der LINQ-Abfrage erfolgt in zwei Varianten:

- verzögerte Ausführung beim Iterieren über die Ergebnismenge mittels *For Each*-Schleife,
- unverzögerte Ausführung durch Anwendung der Konvertierungsmethode *ToArray*.

Oberfläche

Ein Formular, eine *ListBox* und zwei Schaltflächen (siehe Laufzeitansicht).

Quellcode

Öffnen Sie das Codefenster von *Form1*:

```
Public Class Form1
```

Definieren Sie eine generische Personenliste, die Sie mittels Objektinitialisierer mit einigen Personen füllen, darunter auch mehrfach vorkommende:

```
    Dim personen As New List(Of CPerson)() From {
              New CPerson("Müller", "Kurt"), New CPerson("Meier", "Tino"),
              New CPerson("Schneider", "Fernando"), New CPerson("Müller", "Kurt"),
              New CPerson("Gewinnus", "Thomas"), New CPerson("Müller", "Kurt"),
              New CPerson("Müller", "Max"), New CPerson("Müller", "Kurt"),
              New CPerson("Schneider", "Fernando"), New CPerson("Gewinnus", "Thomas"),
              New CPerson("Doberenz", "Walter"), New CPerson("Doberenz", "Walter")}
```

Variante 1:

```
    Private Sub Button1_Click(sender As Object,
                                        e As EventArgs) Handles Button1.Click
```

Personenduplikate werden mittels *Distinct*-Methode entfernt. Übergabeparameter ist eine Instanz von *IEqualityComparer*, um die Regeln festzulegen, nach denen Duplikate identifiziert werden sollen (Definition von *PersonComparer* siehe weiter unten):

```
        Dim personen1 = personen.Distinct(New PersonComparer())
```

Erst beim Iterieren über die Abfrage wird diese ausgeführt:

```
        For Each person As CPerson In personen1
```

Die resultierenden Einträge werden angezeigt:

```
            ListBox1.Items.Add(person.VorName & "  " & person.NachName)
        Next
    End Sub
```

Variante 2:

```
    Private Sub Button2_Click(sender As Object,
                                          e As EventArgs) Handles Button2.Click
        Dim personen1 = personen.Distinct(New PersonComparer())
```

Die folgende LINQ-Abfrage wird sofort ausgeführt und konvertiert die Nachnamen der von Doppelgängern befreiten Personen in eine Stringliste:

```
        Dim personenNN() As String = (From ps In personen1 Select ps.NachName).ToArray
```

Anzeige:

```
        For Each s As String In personenNN
            ListBox1.Items.Add(s)
        Next
    End Sub
End Class
```

Nun kommen wir zu den beiden Hilfsklassen.

Fügen Sie gleich im Anschluss den Code der Klasse *CPerson* ein:

```
Public  Class CPerson
```

Ein eigener Konstruktor:

```
    Public Sub New(nn As String, vn As String)
```

```
        VorName = vn
        NachName = nn
    End Sub
```

Zwei selbstimplementierende Eigenschaften *VorName* und *NachName*:

```
    Public Property VorName As String
    Public Property NachName As String
End Class
```

In einer extra Klasse *PersonComparer* müssen wir nun noch die Regeln zum Vergleich zweier Personen definieren. Die geerbte generische Schnittstelle *IEqualityComparer* erzwingt die Implementierung der Methoden *Equals* und *GetHashCode*:

```
Public Class PersonComparer
    Implements IEqualityComparer(Of CPerson)
```

Die *Equals*-Methode liefert *True*, wenn beide *CPerson*-Objekte gleiche Eigenschaften haben:

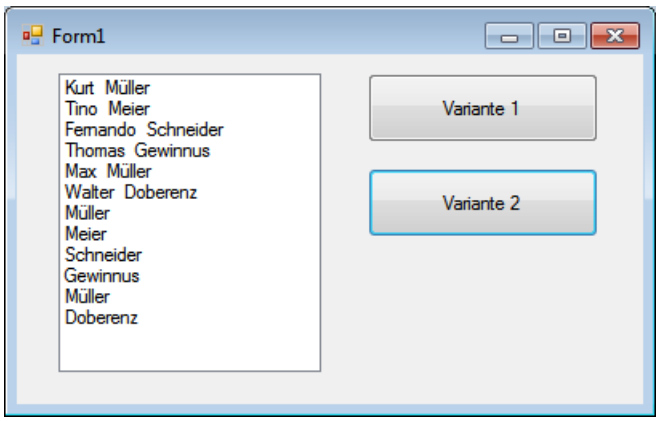
```
    Public Function Equals(first As CPerson, second As CPerson) As Boolean _
                                          Implements IEqualityComparer(Of CPerson).Equals
        Return first.VorName = second.VorName And first.NachName = second.NachName
    End Function
```

GetHashCode liefert einen Integer zwecks eindeutiger Identifikation des Objekts:

```
     Public Function GetHashCode(person As CPerson) As Integer _
                     Implements IEqualityComparer(Of CPerson).GetHashCode
        Return person.NachName.GetHashCode()
    End Function
End Class
```

Test

Die Abfrageergebnisse zeigen jede Person nur einmal (unique). Bei Variante 2 werden allerdings nur die Nachnamen angezeigt (deshalb zweimal "Müller"!).



Bemerkungen

- Bei Variante 1 handelt es sich um das LINQ-typische "verzögerte Ausführen von Abfragen" (*deferred query execution*).
- Die LINQ-Erweiterungsmethode *Distinct* liefert ein Abfrageergebnis vom Typ *IEnumerable(Of CPerson)*, welches mit den Konverierungsmethoden *ToArray*, *ToList* oder *ToDictionary* unverzögert in ein Array oder eine Liste verwandelt werden kann.

6.4.6 Arrays mit LINQ initialisieren

Es kommt häufig vor, dass Sie die Elemente eines Arrays initialisieren müssen, entweder mit demselben Wert oder mit einer sich stetig erhöhenden Folge von Werten. Mittels LINQ können Sie all dies bereits im Array-Initialisierer erledigen – Sie brauchen dazu keine *For*-Schleife!

Die Klasse *Enumerable* aus dem Namespace *System.Linq* stellt dazu statische Methoden zum Abfragen von Objekten bereit, die *System.Collection.Generic.IEnumerable(Of T)* implementieren.

Als Beispiel werden in der folgenden Codezeile alle Elemente des Arrays *a* der Länge 10 mit dem Wert -1 initialisiert, die zweite Zeile initialisiert *b* mit (0,1,..9) und die dritte Zeile initialisiert *c* mit (100,110,…,190):

```
Dim a() As Integer = Enumerable.Repeat(-1, 10).ToArray()
Dim b() As Integer = Enumerable.Range(0, 10).ToArray()
Dim c() As Integer = Enumerable.Range(0, 10).Select(Function(i) 100 + 10 * i).ToArray()
```

Oberfläche

Zum Testen öffnen Sie eine neue WPF-Anwendung und platzieren auf dem Stammelement (*Grid*) drei *ListBox*en und drei *Button*s (siehe Laufzeitabbildung). Weisen Sie im XAML-Code den drei *ListBox*en eine *Name*-Eigenschaft (*lbA*, *lbB*, *lbC*) zu und erzeugen Sie (durch Doppelklick auf jeden der drei *Button*s) drei Eventhandler.

```
...
    <Grid Margin="0,0,2,0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <DockPanel Grid.Column="0" Margin="20">
            <Button Content="A" Click="Button_Click_1" DockPanel.Dock="Top" />
            <ListBox Name="lbA" />
        </DockPanel>
        <DockPanel Grid.Column="1" Margin="20">
            <Button Content="B" Click="Button_Click_2" DockPanel.Dock="Top" />
            <ListBox Name="lbB" />
        </DockPanel>
```

```
        <DockPanel Grid.Column="2" Margin="20">
            <Button Content="C" Click="Button_Click_3" DockPanel.Dock="Top" />
            <ListBox Name="lbC" />
        </DockPanel>
    </Grid>
...
```

Quellcode (MainWindow.xaml.vb)

Hier im Codebehind implementieren Sie die Interaktionslogik:

```
Class MainWindow
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
```

Alle Elemente des Arrays *a* mit dem Wert -1 initialisieren:

```
        Dim a() As Integer = Enumerable.Repeat(-1, 10).ToArray()
```

Anzeige:

```
        For i As Integer = 0 To 9
            lbA.Items.Add(a(i).ToString)
        Next i
    End Sub

    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
```

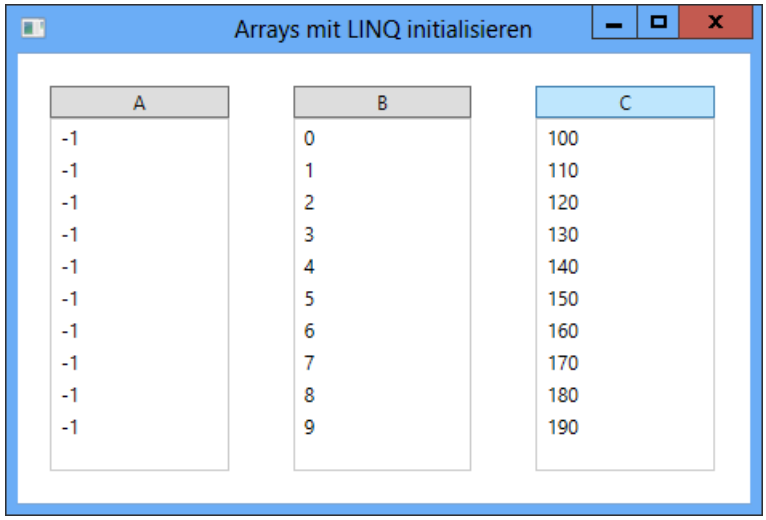
Das Array *b* mit (0, 1, ...9) initialisieren und anzeigen:

```
        Dim b() As Integer = Enumerable.Range(0, 10).ToArray()
        For i As Integer = 0 To 9
            lbB.Items.Add(b(i).ToString)
        Next i
    End Sub
    Private Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
```

Das Array *c* mit (100, 110,…, 190) initialisieren und anzeigen:

```
        Dim c() As Integer = Enumerable.Range(0, 10).Select(Function(i) 100 + 10 * i).ToArray()
        For i As Integer = 0 To 9
            lbC.Items.Add(c(i).ToString)
        Next i
    End Sub

End Class
```

Test



Bemerkungen

Vorsicht – wenn Sie große Arrays initialisieren, sollten Sie doch besser wieder die altmodische *For*-Schleife verwenden. Die LINQ-Lösung vergrößert das Array dynamisch, sodass der Garbage Collector die Arrays zur Laufzeit einsammeln muss. Nutzen Sie deshalb die hier gezeigte Lösung vor allem bei kleineren Arrays oder innerhalb von Test- bzw. Debug-Code.

6.4.7 Arrays per LINQ mit Zufallszahlen füllen

Mit diesem einfachen Trick können Sie eine Zufallsfolge der Länge N generieren:

```
Dim rand As New Random()
Dim randomSeq = Enumerable.Repeat(0, N).Select(Function(i) rand.Next())
```

Oberfläche

Zum Testen öffnen Sie eine neue WPF-Anwendung und erweitern das Formular wie folgt:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Zufallszahlen" Height="350" Width="525">
    <Grid>
        <ListBox Name="ListBox1" ItemsSource="{Binding}"/>
    </Grid>
</Window>
```

Quellcode (MainWindow.xaml.vb)

```
Class MainWindow
```

Fügen Sie folgenden Konstruktor-Code ein:

```
    Public Sub New()
        InitializeComponent()
```

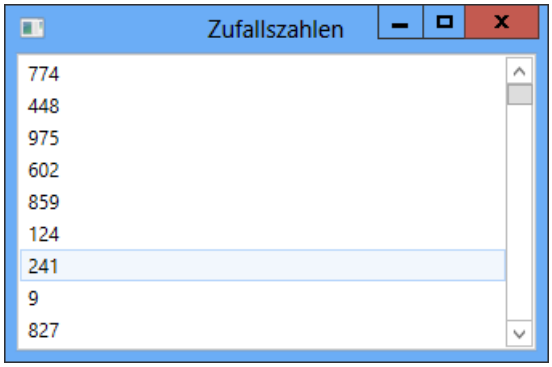
Wir wollen eine Zufallsfolge der Länge 100 erzeugen, die Zufallszahlen zwischen 0 ... 999 generiert:

```
        Dim rand AS New Random()
        Dim randomSeq = Enumerable.Repeat(0, 100).Select(Function(i) rand.Next(0,1000))
```

Dank der verzögerten Ausführung von LINQ-Abfragen wird diese Zufallssequenz nicht sofort berechnet und im Array abgelegt, sondern erst beim Iterieren über die Sequenz, d.h. bei der Anzeige:

```
        ListBox1.DataContext = randomSeq
    End Sub
End Class
```

Test



6.4.8 Einen String mit Wiederholmuster erzeugen

LINQ ist auch hervorragend geeignet, um verschiedenste Arten von Strings zu erzeugen, so zum Beispiel auch Strings für Test- und Debugging-Zwecke.

Wollen Sie zum Beispiel einen String mit dem sich wiederholenden Muster "ABCABCABC…" der Länge N generieren, so gibt es zum Beispiel die folgende elegante Lösung:

```
Dim values As String = String.Join(String.Empty,
Enumerable.Repeat("ABC", N).
ToArray())
```

Oberfläche

Zum Testen öffnen Sie eine neue WPF-Anwendung und platzieren auf dem Stammelement (*Grid*) ein *Label* und einen *Button*. Weisen Sie im XAML-Code dem *Label* eine *Name*-Eigenschaft (*Label1*) zu und erzeugen Sie durch Doppelklick auf den *Button*s einen Eventhandler:

```
...
    <Grid Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="3*"/>
        </Grid.RowDefinitions>
        <Label Name="Label1" Content="Label" HorizontalAlignment="Center"
               VerticalAlignment="Center" />
        <StackPanel Grid.Row="1" Orientation="Horizontal" HorizontalAlignment="Center"
                    Height="50">
            <Button Content="START" Click="Button_Click_1" Width="100" />
        </StackPanel>
    </Grid>
...
```

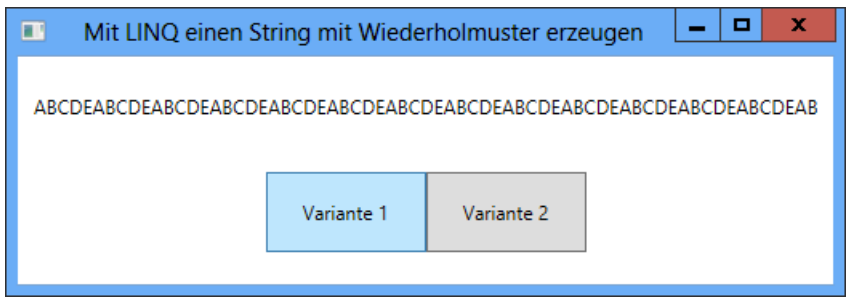
Quellcode (MainWindow.xaml.vb)

Hier im Codebehind implementieren Sie die Interaktionslogik:

```
Class MainWindow

  Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
      Dim values As String = String.Join(String.Empty, Enumerable.Repeat("ABC", 100).ToArray())
      Label1.Content = values
  End Sub

End Class
```

Test

Insgesamt ist der erzeugte String 100 Zeichen lang:



6.4.9 Mit LINQ Zahlen und Strings sortieren

Vorbei ist die Zeit, in der Sie Sortieralgorithmen mühsam per Hand programmieren mussten.So-wohl das Sortieren von Arrays aus Zahlen oder Zeichenketten ist mittels LINQ kein Problem, wie es unser Beispiel zeigen wird.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und fügen Sie im Designer zwei *ListBox*-Controls in das Stammelement ein. Geben Sie den *ListBox*-Controls die Namen *lb1* und *lb2*:

```
...
    <Grid Margin="20">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1*"/>
            <ColumnDefinition Width="2*"/>
        </Grid.ColumnDefinitions>
        <ListBox Name="lb1" Margin="5" Grid.Column="0" ItemsSource="{Binding}"/>
        <ListBox Name="lb2" Margin="5" Grid.Column="1" ItemsSource="{Binding}"/>
    </Grid>
...
```

Quellcode (MainWindow.xaml.vb)

Im Codebehind fügen wir einfach Konstruktorcode hinzu:

```
Class MainWindow()

    Public Sub New()
        InitializeComponent()
```

Eine Liste mit Zahlen:

```
            Dim a() As Integer = {10, -4, 8, 12}
```

Sortieren mit LINQ (standardmäßig aufsteigend) und Anzeige:

```
            a = (From z In a Order By z Select z).ToArray()
            lb1.DataContext = a
```

Irgendeine String-Liste:

```
           Dim namen() As String = {"Mayer", "Zacharias", "Baumann", "Förster", "Abraham"}
```

Zur Abwechslung wird hier in umgekehrter alphabetischer Reihenfolge sortiert:

```
          namen = (From n In namen Order By n Descending Select n).ToArray()
```

Anzeige:

```
          lb2.DataContext = namen
       End Sub
End Class
```

Test

Sofort nach Starten der Anwendung ist das Ergebnis sichtbar:

6.4.10 Mit LINQ Collections von Objekten sortieren

Insbesondere bei der Datenbindung unter WPF haben Sie es häufig mit Objektlisten zu tun. Dabei ist neben dem Abfragen auch das Sortieren eine häufige Aufgabenstellung, die sich mit LINQ elegant erledigen lässt.

In unserem Beispiel wollen wir eine generische Collection sortieren, die aus *Studenten*-Objekten besteht.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie auf dem Stammelement eine *ListBox* (*Name=lb*) und drei Schaltflächen:

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <ListBox Name="lb"  Grid.Column="0" ItemsSource="{Binding}"/>
    <StackPanel Grid.Column="1">
        <Button Content="unverzögert" Click="Button_Click_1" />
        <Button Content="verzögert"  Click="Button_Click_2" />
        <Button Content="Liste löschen" Click="Button_Click_3" />
    </StackPanel>
</Grid>
```

Klasse CStudent

Fügen Sie zum Projekt eine Klasse *CStudent* hinzu:

```
Public Class CStudent
    Public Property Nummer As Integer
    Public Property Vorname As String
```

```
    Public Property Nachname As String

    Public Sub New(numm As Integer, vor As String, nach As String)
        Nummer = numm
        Vorname = vor
        Nachname = nach
    End Sub
```

Das Überschreiben der *ToString*-Methode vereinfacht hier die spätere Datenbindung:

```
    Public Overrides Function ToString() As String
        Return Nummer.ToString() & " ,  " & Nachname & " , " & Vorname
    End Function

End Class
```

MainWindow.xaml.vb

Im Codebehind erzeugen Sie zunächst eine Studentenliste und initialisieren diese mit einigen Einträgen:

```
Class MainWindow

    Private Function getStudentenListe() As CStudent()
        Dim studenten = New CStudent() {New CStudent(82495, "Horst", "Borchert"),
                                        New CStudent(20935, "Gerlinde", "Gräfe"),
                                        New CStudent(79274, "Martha", "Mittag"),
                                        New CStudent(79204, "Emil", "Schmidt"),
                                        New CStudent(92804, "Max", "Pressmann"),
                                        New CStudent(92804, "Heinz", "Hammer")}
        Return studenten
    End Function
```

Die Ausführung (verzögert):

```
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Dim students = getStudentenListe()
```

Die LINQ-Abfrage (die Eigenschaft, nach der sortiert wird, ist hier die *Nummer*):

```
        Dim linqExpr = From studs
                       In students
                       Order By studs.Nummer
                       Select studs
```

Die Ausgabe der Abfrageergebnisse:

```
        lb.DataContext = linqExpr
    End Sub
```

Die sofortige Ausführung (beim Abrufen des Arrays):

```
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Dim studenten = getStudentenListe()
```

Die LINQ-Abfrage:

```
        Dim linqExpr As IEnumerable(Of CStudent) = From studs
                        In students
                        Order By studs.Nachname
                        Select studs
```

Die Abfrage wird sofort ausgeführt:

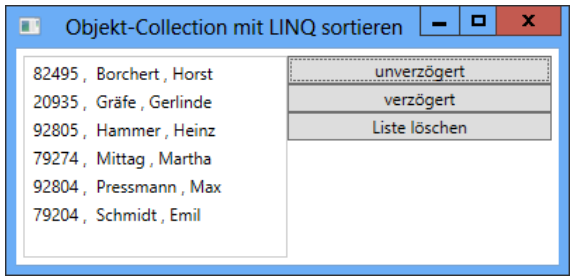
```
        Dim studListe() As CStudent = linqExpr.ToArray()
```

Die Anzeige des Array-Inhalts:

```
        lb.DataContext = studListe
    End Sub
    ...
End Class
```

Test

Die Studentenliste ist in aufsteigender Reihenfolge der Nummern sortiert (siehe folgende Abbildung).



Bemerkungen

- Da die Abfrage normalerweise erst dann ausgeführt wird, wenn auf das Abfrageergebnis (hier *studs*) zugegriffen wird (z.B. per Datenbindung), spricht man deshalb von einer verzögerten Ausführung (*deferred query execution*).
- Natürlich kann LINQ noch viel mehr als nur sortieren. Fügen Sie eine *Where*-Klausel hinzu, um beispielsweise alle Studenten, deren Nummern größer als 80000 sind, herauszufiltern!

6.4.11 Ergebnisse von LINQ-Abfragen in ein Array kopieren

Eine LINQ-Abfrage produziert eine Werteliste, die in einer Variablen gespeichert wird, die innerhalb einer *From*-Klausel erzeugt wird. Falls Sie es wünschen, können Sie das Ergebnis der LINQ-Abfrage aber auch in einer spezifischen Variablen speichern. Das generische *IEnumerable*-Interface stellt dazu die Methoden *ToArray()* bereit. Siehe dazu das Beispiel

► 6.4.10 Mit LINQ Collections von Objekten sortieren

Teil II

Teil II: Technologien

- **Zugriff auf das Dateisystem**
- **Dateien lesen und schreiben**
- **Asynchrone Programmierung**
- **Die Task Parallel Library**
- **Fehlersuche und Behandlung**
- **XML in Therorie und Praxis**
- **Einführung in ADO**
- **Das DataSet**
- **Verteilen von Anwendungen**
- **Weitere Techniken**
- **Konsolenanwendungen**

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 Verzeichnis- und Dateioperationen

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. Weitere interessante Klassen des *System.IO*-Namespace entnehmen Sie der folgenden Tabelle.

Übersicht

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.
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 den **statischen** Klassen *File*, *Directory* und *Path* 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.

BEISPIEL 7.1: Zwei alternative Möglichkeiten zum Anzeigen von Erstellungsdatum und Zeitpunkt des letzten Zugriffs auf die Datei *c:|test|info.txt*

VB

```
Imports System.IO
```

Mit statischer Klasse:

```
Label1.Text = File.GetCreationTime("c:\test\info.txt").ToString
Label2.Text = File.GetLastAccessTime("c:\test\info.txt").ToString
```

BEISPIEL 7.1: Zwei alternative Möglichkeiten zum Anzeigen von Erstellungsdatum und Zeitpunkt des letzten Zugriffs auf die Datei *c:|test|info.txt*

Oder mit Instanzen-Klasse:

```
Dim myFile As New FileInfo("c:\test\info.txt")
Label1.Text = myFile.CreationTime.ToString
Label2.Text = myFile.LastAccessTime.ToString
```

Ein weiterer wichtiger Unterschied zwischen beiden Klassen soll keinesfalls verschwiegen werden:

HINWEIS: Wenn Sie mit den Methoden der statischen Klassen *File*, *Directory* und *Path* arbeiten, werden Sicherheitsüberprüfungen bei **jedem** Methodenaufruf vorgenommen, bei den Instanzen-Methoden der Klassen *FileInfo* und *DirectoryInfo* geschieht dies nur ein einziges Mal.

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
EnumerateFiles	... gibt eine Auflistung von Dateinamen zurück

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	*Sub*	... öffnet die Datei und hängt eine Zeichenfolge an
AppendText	*StreamWriter*	... hängt Text an eine vorhandene Datei an
Copy	*Sub*	... kopiert eine Datei an einen anderen Speicherort
Create	*FileStream*	... erzeugt eine Datei in einem bestimmten Pfad
CreateText	*StreamWriter*	... erzeugt und öffnet eine Textdatei
Delete	*Sub*	... 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	*Sub*	... 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	*Sub*	... setzt die Dateiattribute
SetCreationTime *SetLastAccesTime* *SetLastWriteTime*	*Sub*	... setzt Zeitpunkt der Erstellung, ... des letzten Zugriffs, ... des letzten Schreibzugriffs
WriteAllBytes	*Sub*	... erzeugt neue Datei und schreibt Byte-Array hinein
WriteAllLines	*Sub*	... erzeugt neue Datei und schreibt String-Array hinein
WriteAllText	*Sub*	... 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	*Sub*	... löscht eine Datei
Exists	*Boolean*	... prüft das Vorhandensein einer Datei
MoveTo	*Sub*	... 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

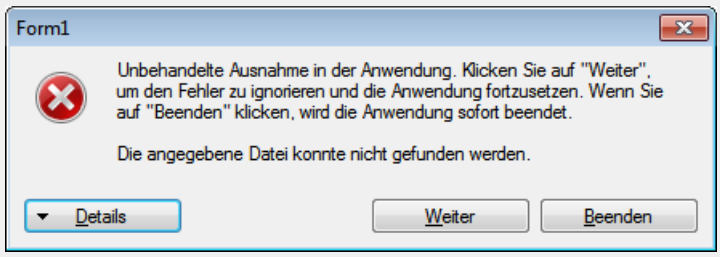
Für diese häufig zu lösende Aufgabenstellung eignet sich die statische *Exists*-Methode der *Directory*- bzw. *File*-Klasse (auch *DirectoryInfo*- und *FileInfo*-Objekte verfügen über diese Methode):

BEISPIEL 7.2: Die Existenz einer Datei feststellen

VB
```
Dim pfad As String = "c:\TestXY.txt"
If Not File.Exists(pfad)) Then 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

Wie es für viele Dateioperationen typisch ist, haben Sie auch hier die Qual der Wahl zwischen zwei Klassen.

Mit Directory-Klasse

Die einfachsten Möglichkeiten zum Erzeugen und Löschen von Verzeichnissen bieten die statischen Methoden *CreateDirectory* und *Delete* der *Directory*-Klasse.

BEISPIEL 7.3: Ein Verzeichnis erzeugen und anschließend wieder löschen

VB

```
Imports System.IO
...
Dim pfad As String = "C:\Temp"
Directory.CreateDirectory(pfad)      ' falls Verzeichnis bereits vorhanden, passiert nichts!
Directory.Delete(pfad, True)         ' löscht auch vorhandene Unterverzeichnisse und Dateien
```

Mit DirectoryInfo-Klasse

Das gleiche Ziel, allerdings etwas umständlicher, erreicht man mit der *Create*-Methode der instanziierbaren *DirectoryInfo*-Klasse, wobei mittels *CreateSubdirectory* auch das Hinzufügen von Unterverzeichnissen möglich ist.

BEISPIEL 7.4: Ein Verzeichnis und ein Unterverzeichnis anlegen und wieder löschen

VB

```
Imports System.IO
...
Dim di As 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 funktioniert nur, wenn das Verzeichnis leer ist!

7.3.3 Verzeichnisse verschieben und umbenennen

Für diese Aufgaben verwenden Sie am besten die *Move*-Methode der statischen *Directory*-Klasse.

BEISPIEL 7.5: Verzeichnis *C:|TempX* wird nach *C:|Beispiele* verschoben und umbenannt in *TempY*

VB
```
Imports System.IO
...
Directory.Move("C:\TempX", "C:\Beispiele\TempY")
```

HINWEIS: Leider gibt sowohl in der *Directory*- als auch in der *DirectoryInfo*-Klasse noch keine Methode, die das Kopieren eines kompletten Verzeichnisses ermöglicht.

7.3.4 Aktuelles Verzeichnis bestimmen

Verwenden Sie dazu die *GetCurrentDirectory*- bzw. *SetCurrentDirectory*-Methode der (statischen) *Directory*-Klasse.

BEISPIEL 7.6: Festlegen und Anzeigen des aktuellen Arbeitsverzeichnisses

VB
```
Imports System.IO
...
Directory.SetCurrentDirectory("C:\Test")
Label1.Text = Directory.GetCurrentDirectory()     ' zeigt "C:\Test"
```

HINWEIS: 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

VB
```
Imports System.IO
...
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

VB
```
Imports System.IO
...
Dim myDir As New DirectoryInfo("C:\")      ' neues DirectoryInfo-Objekt
Dim myDirs() As DirectoryInfo              ' Array zum Speichern der Unterverzeichnisse

myDirs = myDir.GetDirectories()         ' Unterverzeichnisse ermitteln und im Array ablegen
For i As Integer = 0 To  myDirs.Length-1   ' alle Unterverzeichnisse durchlaufen ...
    ListBox1.Items.Add(myDirs(i).Name)     ' ... und Verzeichnisnamen zur ListBox hinzufügen
Next i
```

Eine alternative Lösung bietet sich mit der gleichnamigen Methode der (statischen) *Directory*-Klasse:

BEISPIEL 7.9: Lösung des gleichen Problems wie im Vorgängerbeispiel

VB
```
Imports System.IO
...
Dim myDirs() As String = Directory.GetDirectories("C:\")
For i As Integer = 0 To myDirs.Length-1
    ListBox1.Items.Add(myDirs(i))
Next i
```

7.3.6 Alle Laufwerke ermitteln

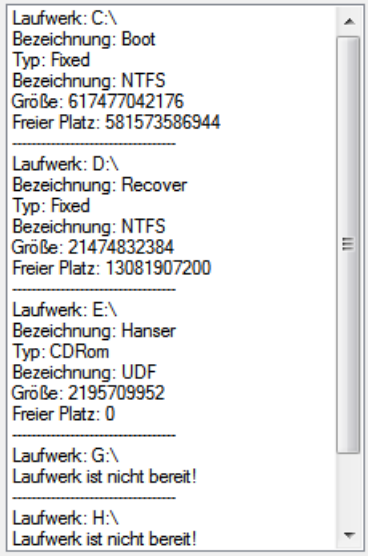
Die Klasse *DriveInfo* ermöglicht den Zugriff auf diverse Laufwerksinformationen. Alles Weitere soll direkt am Beispiel erklärt werden.

BEISPIEL 7.10: Auflisten aller Laufwerke des Systems inklusive dazugehöriger Informationen

VB
```
For Each di As System.IO.DriveInfo In System.IO.DriveInfo.GetDrives()
    With ListBox1.Items
        .Add("Laufwerk: " & di.Name)
        If di.IsReady Then
            .Add("Bezeichnung: " & di.VolumeLabel)
            .Add("Typ: " & di.DriveType)
            .Add("Bezeichnung: " & di.DriveFormat)
            .Add("Größe: " & di.TotalSize)
            .Add("Freier Platz: " & di.TotalFreeSpace)
            .Add("--------------------------------")
        Else
            .Add("Laufwerk ist nicht bereit!")
            .Add("--------------------------------")
        End If
    End With
Next di
```

BEISPIEL 7.10: Auflisten aller Laufwerke des Systems inklusive dazugehöriger Informationen

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

VB

```
Imports System.IO
...
Dim sourcePath As String = "C:\Sample.txt"
Dim destPath As String = "C:\Sample1.txt"
Dim movePath As String = "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

VB

```
Imports System.IO
...
Dim sourcePath As String = "C:\Sample.txt"
Dim destPath As String = "C:\Sample1.txt"
Dim movePath As String = "C:\Temp\Sample1.txt"
Dim myFile As New FileInfo(sourcePath)
```

BEISPIEL 7.12: Obiges Beispiel wird mit Methoden der *FileInfo*-Klasse realisiert

```
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 zum Umbenennen also die Methoden *Move* der Klasse *File* bzw. *MoveTo* der Klasse *FileInfo*.

BEISPIEL 7.13: Umbenennen der Datei *info.txt* in *info_1.txt*

VB
```
Imports System.IO
...
Dim myFile As 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
LastWriteTime	*GetLastWriteTime()* *SetLastWriteTime()*	Datum/Uhrzeit des letzten Schreibzugriffs
Exists	*Exists()*	Liefert *True*, falls Datei physikalisch existiert

BEISPIEL 7.14: Anzeige des Erstellungsdatums einer Datei

VB
```
Imports System.IO
...
Label1.Text = File.GetCreationTime("Liesmich.txt").ToString()
```

BEISPIEL 7.15: Feststellen, ob Datei im Arbeitsverzeichnis existiert

VB
```
Imports System.IO
...
If File.Exists("Liesmich.txt") Then MessageBox.Show("Datei ist vorhanden!")
```

oder

```
Dim myFile As New FileInfo("Liesmich.txt")
If myFile.Exists Then MessageBox.Show("Datei ist vorhanden!")
```

Die FileAttribute-Enumeration

Die verschiedenen Attribute für Dateien und Verzeichnisse sind in der *FileAttribute*-Enumeration anzutreffen. Die folgende Tabelle zeigt die wichtigsten:

Mitglied	Beschreibung
Archive	Entspricht dem Archiv-Status der Datei, wie er häufig zum Markieren einer zu löschenden oder einer Backup-Datei verwendet wird
Compressed	Entspricht einer gepackten Datei
Directory	Zeigt an, dass die Datei in Wirklichkeit ein Verzeichnis ist
Encrypted	Die Datei ist verschlüsselt
Hidden	Die Datei ist versteckt und demzufolge 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

VB
```
Dim attbs As FileAttributes = File.GetAttributes("C:\Beispiele\Test.dat")
If attbs = (attbs Or FileAttributes.Archive) Then
    CheckBox2.Checked = True
Else
    CheckBox2.Checked = False
End If
```

7.3.10 Verzeichnis einer Datei ermitteln

BEISPIEL 7.17: Das Verzeichnis der Datei *liesmich.txt* wird in einem Label angezeigt

VB
```
Dim myFile As New FileInfo("Liesmich.txt")
Label1.Text = myFile.DirectoryName
```

7.3.11 Alle im Verzeichnis enthaltene 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 Rootverzeichnis *c:|* abgelegten Dateien werden in einer *ListBox* angezeigt

VB
```
Dim myDir As New DirectoryInfo("c:\")          ' neues DirectoryInfo-Objekt
Dim myFiles() As FileInfo = myDir.GetFiles()   ' Dateien ermitteln und im Array ablegen

For i As Integer = 0 To myFiles.Length - 1     ' alle Dateien durchlaufen ...
     ListBox1.Items.Add(myFiles(i).Name)       ' ... und Dateinamen zur ListBox hinzufügen
Next i
```

Noch kürzer ist der Code bei Verwendung der (statischen) *Directory*-Klasse.

BEISPIEL 7.19: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst

VB
```
Dim myFiles() As String = Directory.GetFiles("c:\")    ' String-Array füllen

For i As Integer = 0 To myFiles.Length -1              ' alle Einträge durchlaufen
     ListBox1.Items.Add(myFiles(i))                    ' ... und anzeigen
Next i
```

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

VB
```
Dim topDir As String = "c:\doberenz"
Dim myFiles() AS String = Directory.GetFiles(topDir, "*.htm", SearchOption.AllDirectories)
For I As Integer = 0 To myFiles.Length - 1       ' alle Einträge durchlaufen
     ListBox1.Items.Add(myFiles(i))              ' ... und anzeigen
Next i
```

7.3.12 Dateien und Unterverzeichnisse ermitteln

Im Zusammenhang mit der *Directory*-Eigenschaft der *FileInfo*-Klasse verdient die *GetFileSystemInfos*-Methode der *DirectoryInfo*-Klasse besondere Aufmerksamkeit.

BEISPIEL 7.21: In einer *TextBox* (*MultiLine* = *True*) werden neben dem Verzeichnis einer Datei alle weiteren sich im gleichen Verzeichnis befindlichen Dateien und Unterverzeichnisse angezeigt.

VB
```
Imports System.IO
...
Dim myFile As New FileInfo("Test.txt")  ' öffnet oder erzeugt Datei im Anwendungsverzeichnis
Dim myDir As DirectoryInfo = myFile.Directory          ' Verweis auf Verzeichnis erzeugen
Dim fsi() As FileSystemInfo = myDir.GetFileSystemInfos()     ' alle Einträge ermitteln
TextBox1.Text = myDir.FullName & Environment.NewLine         ' vollständigen Pfad anzeigen
For Each info As FileSystemInfo In fsi    ' ... alle weiteren Unterverzeichnisse und Dateien
    TextBox1.Text &= info.Name & Environment.NewLine
Next info
```

HINWEIS: Wer seine Erkundungen zu Datei- und Verzeichnisoperationen mit einem komplexeren Beispiel beenden möchte, dem sei das PB 7.7.1 "Infos über Verzeichnisse und Dateien gewinnen" nahe gelegt, an welchem er viele der in diesem Abschnitt besprochenen Features der *FileInfo*-Klasse noch einmal in ihrem 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

VB

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.

```
Imports System.IO
Imports System.Security.AccessControl
...
Dim dir As String = "Testverzeichnis"
Directory.CreateDirectory(dir)
Dim account As String = "Jeder"
Dim dsc As DirectorySecurity = Directory.GetAccessControl(dir)
Dim rights As FileSystemRights = FileSystemRights.FullControl
Dim controlType As AccessControlType = AccessControlType.Deny
Dim fsar As 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.

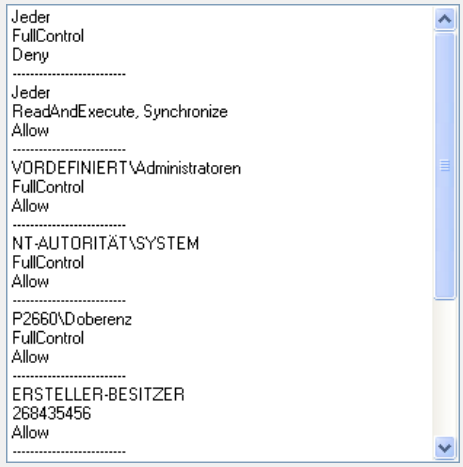
BEISPIEL 7.23: Die ACL des im Vorgängerbeispiel erzeugten Verzeichnisses wird ausgelesen, die einzelnen ACEs werden in einer *ListBox* angezeigt.

VB

```
Imports System.IO
Imports System.Security.AccessControl
...
Dim dir As String = "Testverzeichnis"
Dim fsc As DirectorySecurity = Directory.GetAccessControl(dir)
Dim arcoll As AuthorizationRuleCollection = fsc.GetAccessRules(True, True,
                                    Type.GetType("System.Security.Principal.NTAccount"))
For Each fsar As FileSystemAccessRule In arcoll
    With ListBox1.Items
        .Add(fsar.IdentityReference.ToString())
        .Add(fsar.FileSystemRights.ToString())
        .Add(fsar.AccessControlType.ToString())
        .Add("-------------------------")
    End With
Next
```

BEISPIEL 7.23: Die ACL des im Vorgängerbeispiel erzeugten Verzeichnisses wird ausgelesen, die einzelnen ACEs werden in einer *ListBox* angezeigt.

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 Member 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

Die zahlreichen (statischen) Methoden sollen hier lediglich anhand eines Beispiels demonstriert werden.

[1] ... einschließlich des führenden Punkts

BEISPIEL 7.24: Ausgabe von Dateiinfos in einer *ListBox*

VB
```
Imports System.IO.Path
...
Dim verz As String = "C:\Test\Info.txt"
With ListBox1.Items
    .Add("Verzeichnis : " & GetDirectoryName(verz))
    .Add("Dateiname : " & GetFileName(verz))
    .Add("Dateiname ohne Extension : " & GetFileNameWithoutExtension(verz))
    .Add("Dateiextension : " & GetExtension(verz))
    .Add("Rootverzeichnis : " & GetPathRoot(verz))
    .Add("Temporäres Verzeichnis : " & GetTempPath())
    .Items.Add("Neues Tempfile : " & GetTempFileName())
End With
```

7.5.2 Die Klasse FileSystemWatcher

Die Klasse *FileSystemWatcher* dient dem einfachen Beobachten des Dateisystems. So löst sie 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 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*

VB

```
Private fsWatcher As New System.IO.FileSystemWatcher
```

Alle erforderlichen Eigenschaftszuweisungen und das Anmelden der Ereignisbehandlungen werden bequemlichkeitshalber gleich mit im Konstruktorcode des Formulars erledigt:

```
Public Sub New()
    InitializeComponent()
    With fsWatcher
      .Path = System.IO.Directory.GetCurrentDirectory
      .NotifyFilter = System.IO.NotifyFilters.LastAccess Or System.IO.NotifyFilters.FileName
      .Filter = "*.txt"
      AddHandler .Changed, AddressOf Me.OnChanged     ' Datei wurde geändert
      AddHandler .Created, AddressOf Me.OnChanged     ' Datei wurde neu hinzugefügt
      AddHandler .Deleted, AddressOf Me.OnChanged     ' Datei wurde gelöscht
      AddHandler .Renamed, AddressOf Me.OnRenamed     ' Datei wurde umbenannt
   End With
End Sub
```

Die beiden folgenden Event-Handler spezifizieren die Reaktion auf die vier Ereignisse, die Anzeige erfolgt in einer *ListBox*:

```
Public Sub OnChanged(Source As Object, e As FileSystemEventArgs)
   Try
     MessageBox.Show("Datei: " & e.FullPath & " " & e.ChangeType.ToString,
                     "Meldung vom FileSystemWatcher")
   Catch ex As Exception
      MessageBox.Show(ex.Message, "Fehler")
   End Try
End Sub

Public Sub OnRenamed(Source As Object, e As RenamedEventArgs)
   Try
     MessageBox.Show("Datei: " & e.OldFullPath & " umbenannt in " & e.FullPath,
                                                  "Meldung vom FileSystemWatcher")
   Catch ex As Exception
     MessageBox.Show(ex.Message)
   End Try
End Sub
```

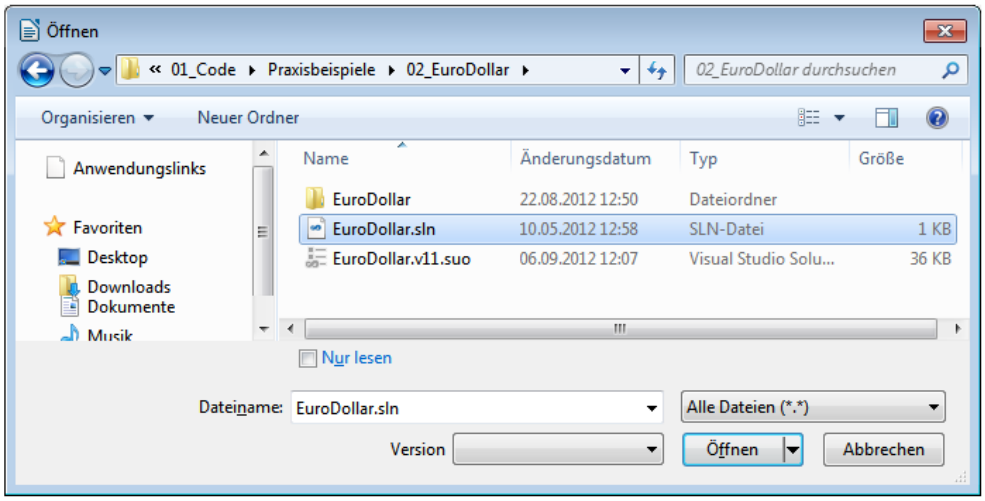
7.6 Datei- und Verzeichnisdialoge

Fast jedes Windows-Programm hat eine Menüoption zum Laden (Öffnen) bzw. Sichern (Speichern) von Dateien. Sollten Sie einen solchen Dateidialog schon einmal "zu Fuß" programmiert haben, so wissen Sie, welcher Aufwand dafür nötig ist. .NET bietet Ihnen dafür die Controls *OpenFileDialog* und *SaveFileDialog*, welche direkt die Windows-Ressourcen "anzapfen". Ihre eigenen Programmschöpfungen erhalten damit die gleichen Dateidialoge 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

VB
```
Dim dateiname As String = String.Empty
If OpenFileDialog1.ShowDialog() =  DialogResult.OK Then
    dateiname = OpenFileDialog1.FileName
    ...
End If
```

BEISPIEL 7.27: Auswahl von mehreren Dateien (*MultiSelect=True*)

VB
```
If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
    For i As Integer = 0 To OpenFileDialog1.FileNames.GetUpperBound(0)
```

BEISPIEL 7.27: Auswahl von mehreren Dateien (*MultiSelect=True*)

```
        MessageBox.Show(OpenFileDialog1.FileNames(i))
    Next
End If
```

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
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 Ursprungs-dateiname 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

VB
```
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)

VB
```
FolderBrowserDialog1.RootFolder = System.Environment.SpecialFolder.UserProfile
Dim result As DialogResult = folderBrowserDialog1.ShowDialog()
If result = DialogResult.OK Then
    startFolder = folderBrowserDialog1.SelectedPath
    Label1.Text = startFolder            ' Anzeige des ausgewählten Verzeichnisses
End If
```

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 zeigt 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 Schluss).

Quelltext

```
Imports System.IO

Public Class Form1
```

Globale Deklarationen auf Form-Ebene:

```
    Private myRoot As String = "C:\"                    ' übergeordnetes Verzeichnis
    Private myDirName, myFileName As String
    Private CrLf As String = Environment.NewLine        ' für Zeilenumbruch in den Labels
```

Die Startaktivitäten können beim Laden des Formulars erledigt werden:

```
   Protected Overrides Sub OnLoad(e As EventArgs)
        TextBox1.Text = myRoot
        showDirectories()
        MyBase.OnLoad(e)
    End Sub
```

Die folgende Methode zeigt alle zu *myRoot* untergeordneten Verzeichnisse in *ListBox1* an:

```
   Private Sub showDirectories()
        Dim myDirectories() As DirectoryInfo   ' Array zum Speichern der Unterverzeichnisse
```

Erzeugen eines neuen *DirectoryInfo*-Objekts, welches auf das Rootverzeichnis zeigt:

```
        Dim myDirectory As New DirectoryInfo(myRoot)
```

Alle Unterverzeichnisse ermitteln und abspeichern (vorher Anzeige löschen):

```
        myDirectories = myDirectory.GetDirectories()
        ListBox1.Items.Clear()
```

Alle Verzeichnisse durchlaufen ...

```
         For i As Integer = 0 To myDirectories.Length - 1
```

... und Verzeichnisnamen zur *ListBox* hinzufügen:

```
             ListBox1.Items.Add(myDirectories(i).Name)
```

Der erste Eintrag in der Verzeichnis-*ListBox* wird selektiert, dadurch wird das *SelectedIndexChanged*-Event ausgelöst:

```
            ListBox1.SelectedIndex = 0
        Next i
```

Synchronisieren aller Files in der Datei-*ListBox* mit dem selektierten Verzeichnis und Anzeige der Verzeichnis-Informationen:

```
    Private Sub ListBox1_SelectedIndexChanged(sender As Object,
                        e As EventArgs) Handles ListBox1.SelectedIndexChanged
        Dim dirInfo As String = String.Empty
        Dim myFiles() As FileInfo           ' Array für alle Dateiinformationen
        ListBox2.Items.Clear()              ' aktuellen Inhalt löschen
```

DirectoryInfo-Objekt aufgrund des selektierten *ListBox*-Eintrags erzeugen:

```
        myDirName = ListBox1.SelectedItem.ToString() & "/"
        Dim myDirectory As New DirectoryInfo(myRoot & myDirName)   ' aktuelles Verzeichnis
```

Verzeichnis-Infos zusammensetzen:

```
        dirInfo &= "Pfad: " & myDirectory.FullName & CrLf
        dirInfo &= "Erstellungsdatum: " & myDirectory.CreationTime.ToString & 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 Then
            For i As Integer = 0 To myFiles.Length - 1
                ListBox2.Items.Add(myFiles(i).Name)
            Next i
```

Ersten Eintrag in der Datei-*ListBox* selektieren, dadurch wird deren *SelectedIndexChanged*-Event von *ListBox2* ausgelöst:

```
            ListBox2.SelectedIndex = 0
        End If
    End Sub

    Private Sub ListBox2_SelectedIndexChanged(sender As Object,
                                e As EventArgs) Handles ListBox2.SelectedIndexChanged
        Dim fileInf As String = String.Empty
```

Dateinamen zuweisen:

```
        myFileName = ListBox2.SelectedItem.ToString()
```

Neues File-Objekt erzeugen:

```
        Dim myFile As 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
    End Sub
```

Nachdem Sie das übergeordnete Verzeichnis editiert haben, kann die Eingabetaste ausgewertet werden, um die Eingabe abzuschließen:

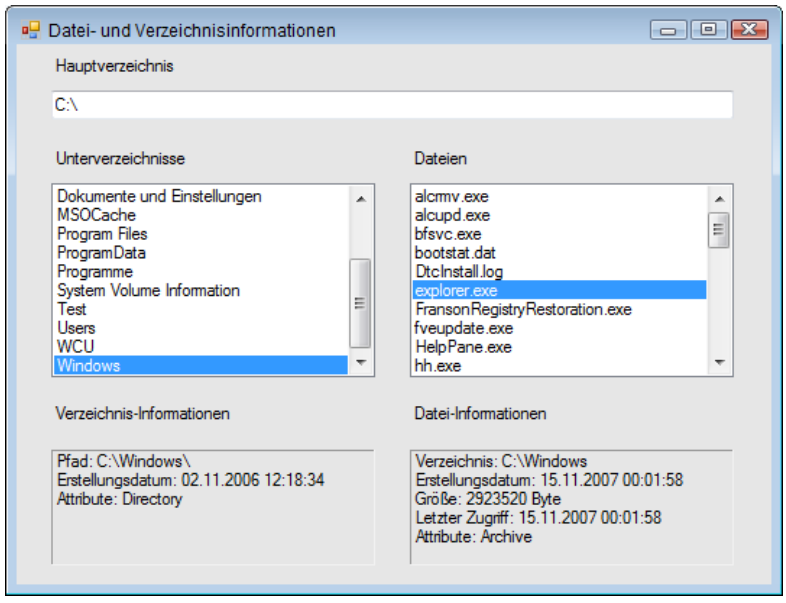
```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter Then
            myRoot = TextBox1.Text
```

Letzten Slash ergänzen, falls notwendig:

```
            If Not myRoot.EndsWith("/") Then myRoot &= "/"
            showDirectories()
        End If
    End Sub
End Class
```

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 weiterbewegen 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 Die Verzeichnisstruktur in eine 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

```
Imports System.IO

Public Class Form1
```

Die Startaktivitäten:

```
    Protected Overrides Sub OnLoad(e As EventArgs)
```

Wurzelknoten erzeugen:

```
        Dim rootNode As New TreeNode("C:\")
        TreeView1.Nodes.Add(rootNode)
```

Untergeordnete Ebene füllen ...

```
        addChildNodes(rootNode)
```

... und expandieren:

```
        TreeView1.Nodes(0).Expand()
        MyBase.OnLoad(e)
    End Sub
```

Die Hauptarbeit erledigt die Methode *addChildNodes*, welcher als Parameter ein Knoten (*dirNode*) übergeben wird. Im Ergebnis werden alle Knoten der untergeordneten Verzeichnisebene hinzugefügt:

```
    Private Sub addChildNodes(dirNode As TreeNode)
        Dim dir As New DirectoryInfo(dirNode.FullPath)
        Try
```

Alle Unterverzeichnisse durchlaufen:

```
            For Each dirItem As DirectoryInfo In dir.GetDirectories()
```

Einen Child-Knoten für jedes Unterverzeichnis hinzufügen:

```
                Dim newNode As 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("*")
            Next dirItem
        Catch err As UnauthorizedAccessException
            MessageBox.Show(err.ToString())
        End Try
    End Sub
```

Ein Knoten wird expandiert (aber die untergeordnete Ebene noch nicht gezeichnet):

```
    Private Sub TreeView1_BeforeExpand(sender As Object, e As TreeViewCancelEventArgs) _
                                                        Handles TreeView1.BeforeExpand
```

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 = "*" Then
```

Erneutes Zeichnen deaktivieren:

```
            TreeView1.BeginUpdate()
```

Platzhalterknoten löschen:

```
            e.Node.Nodes.Clear()
```

Alle untergeordneten Knoten hinzufügen:

```
            addChildNodes(e.Node)
```

Erneutes Zeichnen aktivieren:

```
            TreeView1.EndUpdate()
        End If
    End Sub
```

Die Knoten-Auswahl wurde durch den Anwender geändert:

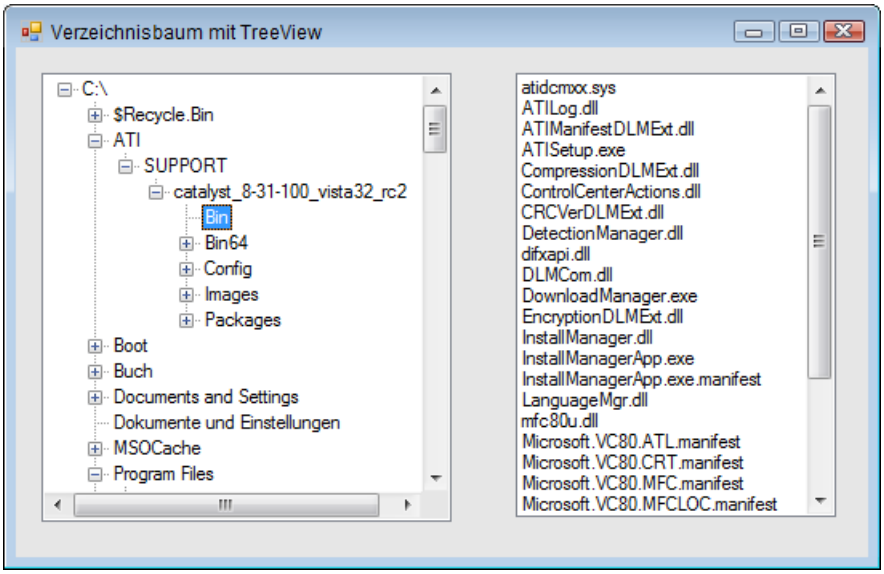
```
    Private Sub TreeView1_AfterSelect(sender As Object, e As TreeViewEventArgs) _
                                                      Handles TreeView1.AfterSelect
```

Alle im entsprechenden Verzeichnis enthaltene Dateien werden in der *ListBox* angezeigt:

```
        Dim dir As New DirectoryInfo(e.Node.FullPath)
        ListBox1.Items.Clear()
        ListBox1.Items.AddRange(dir.GetFiles())
    End Sub
End Class
```

Test

Bewegen Sie sich durch die *TreeView*! Klicken Sie links auf ein bestimmtes Verzeichnis, werden rechts die darin enthaltenen Dateien angezeigt.

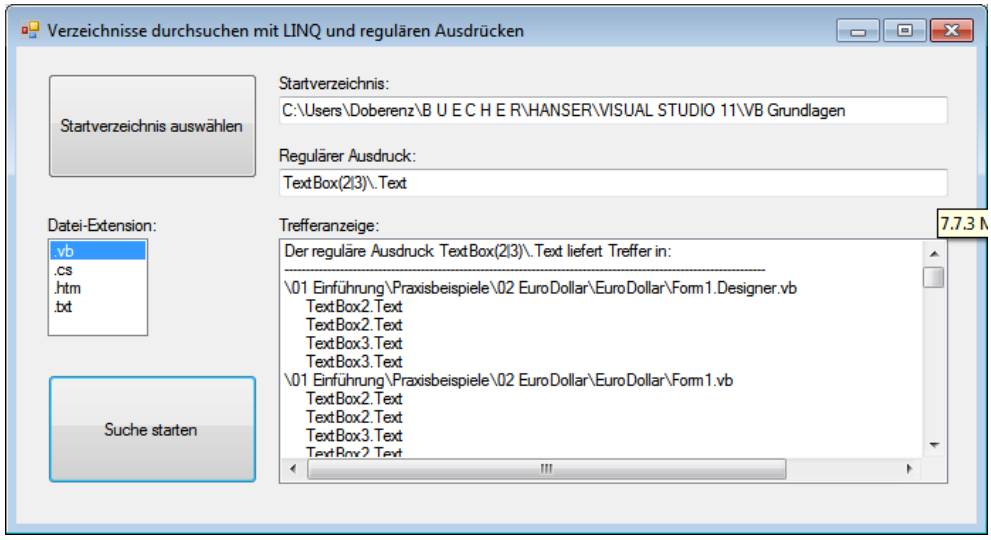


7.7.3 Mit LINQ und RegEx Verzeichnisbäume durchsuchen

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 (siehe Laufzeitansicht).



Zur Auswahl des Startverzeichnissen benötigen wir noch einen *FolderBrowserDialog*. *Die Items*-Auflistung der *ListBox2* (links) füllen wir im Zeichenfolgen-Editor mit verschiedenen Dateiextensions.

Quellcode

```
Imports System.Text.RegularExpressions
Imports System.IO

Public Class Form1
```

Private Variablen für das Startverzeichnis und den gewählten Datei-Typ:

```
    Private startFolder As String = ""
    Private ext As String = ""
```

Das Startverzeichnis auswählen:

```
  Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Anfangseinstellung ist Ihr Benutzer-Verzeichnis:

```
      FolderBrowserDialog1.RootFolder = System.Environment.SpecialFolder.MyComputer
      Dim res As DialogResult = Me.FolderBrowserDialog1.ShowDialog()
      If res = DialogResult.OK Then
        startFolder = Me.FolderBrowserDialog1.SelectedPath
        TextBox1.Text = startFolder
        Button2.Enabled = True
      End If
  End Sub
```

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 Function getAllFiles(path As String) As IEnumerable(Of FileInfo)
```

Überprüfen, ob das Verzeichnis vorhanden ist:

```
    If Not Directory.Exists(path) Then Throw New DirectoryNotFoundException
```

Der Puffer für die Dateinamen:

```
    Dim fileNames As String() = Nothing
```

Die generische Liste:

```
    Dim files As List(Of FileInfo) = New List(Of FileInfo)
```

Alle Dateien in allen Unterverzeichnissen werden ermittelt:

```
    fileNames = Directory.GetFiles(path, "*" & ext, SearchOption.AllDirectories)
    For Each name As String In fileNames
        files.Add(New FileInfo(name))
    Next
    Return files
End Function
```

Die Schaltfläche "Suche starten":

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    ListBox1.Items.Clear()
```

Den regulären Ausdruck zuweisen:

```
    Dim rxPattern As String = TextBox2.Text
    Dim rgx As Regex = New Regex(rxPattern)
```

Datei-Extension zuweisen:

```
    ext = ListBox2.SelectedItem.ToString
```

Alle Dateien in eine generische Liste laden (siehe obige Hilfsmethode):

```
    Dim fList As IEnumerable(Of FileInfo) = getAllFiles(startFolder)
```

Die LINQ-Abfrage erzeugt eine Liste der Dateien, in denen es Treffer gibt, und eine Liste der Treffer in jeder Datei:

```
    Dim qryMatchFiles = From f In fList
```

Eine *Where*-Klausel ist eigentlich überflüssig, da die Dateiliste bereits nach Dateityp gefiltert ist:

```
                        ' Where f.Extension = ext
```

ReadAllText öffnet die Textdatei, liest alle Zeilen und schließt die Datei wieder:

```
                          Let fText = File.ReadAllText(f.FullName)
```

Jetzt erfolgt in der Datei die Suche nach Treffern unseres regulären Ausdrucks:

```
                        Let matches = rgx.Matches(fText)
                        Where (matches.Count > 0)
                        Select Name = f.FullName,
```

Im folgenden LINQ-Ausdruck ist eine explizites Typecasting erforderlich, da *Match* kein Mitglied einer generische *IEnumerable* Collection ist:

```
                          Matches = From match As 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]):

```
    For Each fMatch In qryMatchFiles
```

Pfad der Fundstelle (ohne Startverzeichnis):

```
        Dim s As String = fMatch.Name.Substring(startFolder.Length)
```

Verzeichnisnamen ausgeben:

```
        ListBox1.Items.Add(s)
```

Alle Treffer in dieser Datei ausgeben:

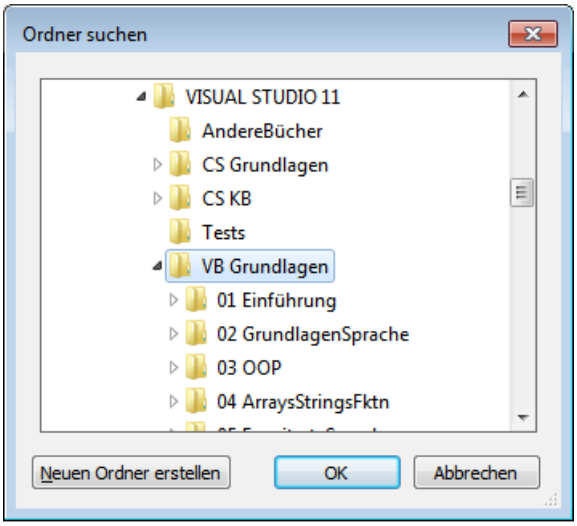
```
        For Each match In fMatch.Matches
            ListBox1.Items.Add("     " & match)
         Next
      Next
  End Sub

End Class
```

Test

Wählen Sie zunächst den Dateityp und dann das Startverzeichnis aus (das ist die Wurzel des zu durchsuchenden Verzeichnisbaums).

HINWEIS: 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 (**.vb*) 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

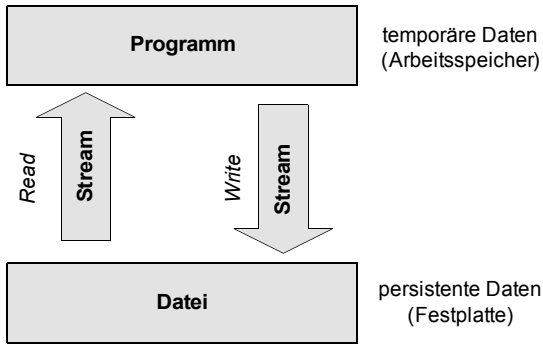
Während ein Programm mit *temporären* bzw. *transienten* Daten arbeitet, die im Arbeitsspeicher abgelegt sind, bezeichnet man die in einer Datei auf Festplatte, CDROM, Stick etc. dauerhaft gespeicherten Informationen als *persistente* Daten.

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 der 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.

VB
```
Imports System.IO
...
Dim myFile As New FileInfo("tmp.txt")
Dim myStream As FileStream = myFile.Open(FileMode.OpenOrCreate, FileAccess.ReadWrite,
                                                                FileShare.None)
```

oder

```
Dim myStream As FileStream = File.Open("tmp.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*).

VB
```
Imports System.IO
...
Dim myStream As 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 im Arbeitsverzeichnis erzeugen und mit dem Inhalt einer *TextBox* füllen.

VB
```
Imports System.IO
...
Dim writer As StreamWriter = File.CreateText("Liesmich.txt")
For Each s As String In TextBox1.Lines
  writer.WriteLine(s)
Next
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.

VB
```
...
Dim pfad As String =  "Liesmich.txt"
Dim file As New FileInfo(pfad)
Dim writer As StreamWriter = file.AppendText()
For i As Integer = 0 To 9
    writer.Write("Hallo")
    writer.Write(Environment.NewLine)
Next i
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*

VB

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.

```
Dim pfad As String = "Liesmich.txt"
If Not File.Exists(pfad) Then
  Dim text() As String = {"Liebe VB-Freunde", "dies hier ist eine Textdatei", "zum Testen "}
  File.WriteAllLines(pfad, text)
End If
```

Datei nochmals öffnen, um eine weitere Textzeile hinzuzufügen:

```
Dim text2 As String = "... 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.

VB
```
Imports System.IO
...
TextBox1.Text = String.Empty
Dim reader As StreamReader = File.OpenText("Liesmich.txt")
While reader.Peek() > -1
     TextBox1.Text &= reader.ReadLine() & Environment.NewLine
End While
reader.Close()
```

Die *ReadToEnd*-Methode vereinfacht den Leseprozess.

BEISPIEL 8.7: Eine alternative Variante des Vorgängerbeispiels

VB
```
...
Dim reader As 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

VB
```
...
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*

VB
```
Dim lines() As String = File.ReadAllLines("Liesmich.txt")
For Each line In lines
 ListBox1.Items.Add(line)
Next 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.

Eine Lösung wäre der Einsatz eines *TextReader* um die Datei zu öffnen und diese zeilenweise in den Speicher einzulesen.

BEISPIEL 8.10: Alternative Lösung des Vorgängerbeispiels, die für größere Textdateien geeigneter ist

VB
```
Using reader As TextReader = New StreamReader("Liesmich.txt")
 Dim line As String = reader.ReadLine()
```

So lange Zeilen lesbar sind ...

```
 Do While line IsNot Nothing
   ListBox1.Items.Add(line)
   line = reader.ReadLine()
 Loop
End Using
```

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(Of 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

VB
```
Dim lines As IEnumerable(Of String) = File.ReadLines(pfad)

For Each line In lines
  ListBox1.Items.Add(line)
Next 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

VB
```
Dim lines As IEnumerable(Of String) = File.ReadLines(pfad)

For Each line In lines
   ListBox1.Items.Add(line)
   If line.Length > 30 Then  Exit For
Next line
```

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].

VB
```
Dim file_A As New FileStream("c:\Bild_A.bmp", FileMode.Open)
Dim reader As New BinaryReader(file_A)
Dim file_B As New FileStream("c:\Bild_B.bmp", FileMode.OpenOrCreate)
Dim writer As New BinaryWriter(file_B)

While reader.PeekChar() > -1
       writer.Write(reader.ReadByte())          ' liest und schreibt ein Byte
End While

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).

VB
```
Dim bytes() As Byte = File.ReadAllBytes("c:\Bild_A.bmp")
File.WriteAllBytes("c:\Bild_B.bmp", bytes)
```

[1] Eine komplette Datei lässt sich natürlich auch durch Anwenden der (statischen) *Copy*-Methode der *File*-Klasse kopieren.

8.4.3 BinaryReader/BinaryWriter erzeugen

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

VB

Variante 1:

```
Dim readStream As New FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Read)
Dim reader As New BinaryReader(readStream)
```

Variante 2:

```
Dim file As New FileInfo(pfad)
Dim reader As New BinaryReader(file.OpenRead())
```

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

VB

Ein Array *pListe*, welches die Daten eines *Structure*-Datentyps (*Nachname*, *Geburtsdatum*, *Student (Ja/Nein)*) speichert, wird in einer sequenziellen Datei zwischengespeichert. Diese Methode schreibt den Arrayinhalt in die Datei:

```
Private Structure Person
    Dim Vorname, NachName As String
    Dim Geburt As Date
    Dim Student As Boolean
End Structure
```

Die Datei befindet sich einfachheitshalber gleich mit im Projektverzeichnis:

```
Private pfad As String = "Personen.dat"
...
```

BEISPIEL 8.16: Lesen und schreiben von strukturierten Daten

Diese Methode schreibt den Arrayinhalt in die Datei:

```
Private Sub writeFile()
      Dim wStream As New FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Write)
      Dim binWriter As New BinaryWriter(wStream)
      For i As Integer = 0 To pmax - 1             ' alle Datensätze durchlaufen
          With pListe(i)
```

Die *Write*-Methode des *BinaryWriter* verfügt über eine Überladung für fast jeden Datentyp:

```
              binWriter.Write(.Vorname)
              binWriter.Write(.Nachname)
              binWriter.Write(.Geburt.ToShortDateString())
              binWriter.Write(.Student)
          End With
      Next i
      binWriter.Flush()   ' Puffer => Disk
      binWriter.Close()
      wStream.Close()
  End Sub
```

Die folgende Methode macht es umgekehrt, sie füllt ein Array mit dem Dateiinhalt:

```
  Private Sub readFile()
      Dim rStream As New FileStream(pfad, FileMode.OpenOrCreate, FileAccess.Read)
      Dim binReader As New BinaryReader(rStream)
      If (rStream.Length > 0) Then            ' nicht bei neu erzeugter Datei
          For i As Integer = 0 To pmax - 1        ' alle Datensätze durchlaufen
              With pListe(i)
```

Der *BinaryReader* verfügt für fast jeden Datentyp über eine spezielle Lesemethode:

```
                  .Vorname = binReader.ReadString()
                  .Nachname = binReader.ReadString()
                  .Geburt = Convert.ToDateTime(binReader.ReadString())
                  .Student = binReader.ReadBoolean()
              End With
          Next i
      End If
      binReader.Close()
      rStream.Close()
  End Sub
```

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 wieder lesen.

VB

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
...
<Serializable> Public Class CKunde            ' Wichtig!

    ...          ' irgendeine Klassenimplementierung

End Class
```

Initialisiertes Objekt erzeugen, serialisieren und in Datei schreiben:

```
Dim kunde As New CKunde(10, Max, Müller, 100.65)
Dim fileOutput As New FileStream("c:\Kunden.dat", FileMode.OpenOrCreate, FileAccess.Write)
Dim binFttr As New BinaryFormatter()
binFttr.Serialize(fileOutput, kunde)
```

Objekt aus Datei zurück lesen und wieder "zusammenbauen" (deserialisieren):

```
Dim kunde As New CKunde()                          ' leeres Record-Objekt erzeugen
Dim fileInput As New FileStream("c:\Kunden.dat", FileMode.Open, FileAccess.Read)
```

Explizite Typumwandlung erforderlich:

```
kunde = CType(binFttr.Deserialize(fileInput), CPerson)
```

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 Einen Objektbaum speichern

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

VB
```
Imports System.IO
...
File.Encrypt("Beispiel.txt")
File.Decrypt("Beispiel.txt")
```

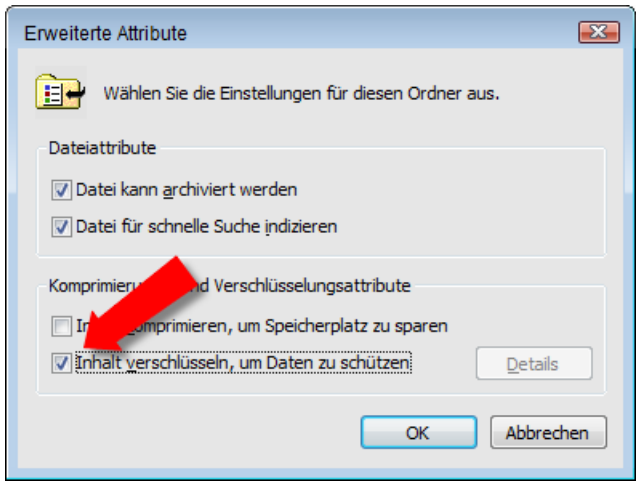
HINWEIS: Erkennen kann man die Verschlüsselung durch die grüne Farbe des Dateinamens und – wenn die Dateiattribute angezeigt werden – durch das E-Attribut (für *Encrypted*).

8.6.2 Verschlüsseln unter Windows Vista/7/8/10

Um eine Datei oder einen Ordner zu verschlüsseln wählen Sie im Kontextmenü den Eintrag *Eigenschaften* und klicken im Eigenschaftenfenster 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:
```
Public Sub New (stream As Stream, transform As ICryptoTransform,
                mode As CryptoStreamMode )
```

Zu den einzelnen Parametern:

- *stream*
  ... 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 S*ystem.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.

VB
```
Imports System.IO
Imports System.Security.Cryptography
...
Private des As New DESCryptoServiceProvider()
Dim desEncrypt As ICryptoTransform = des.CreateEncryptor()
Dim fs As New FileStream("EncryptedFile.txt", FileMode.Create, FileAccess.Write)
Dim cryptoStrm As New CryptoStream(fs, desEncrypt, CryptoStreamMode.Write)
```

BEISPIEL 8.19: Der Inhalt eines Byte-Arrays wird nach dem Data Encryption Standard (DES) verschlüsselt und in der Datei *EncryptedFile.txt* abgelegt.

```
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: Mit der *GZipStream*-Klasse wird der Inhalt einer Datei gepackt und entpackt[1].

VB

```
Imports System.IO
Imports System.IO.Compression

Public Class Form1
    Private QuellDatei As String = "Test1.txt"
    Private KompDatei As String = "Test.zip"
    Private ZielDatei As String = "Test2.txt"
```

Ein Puffer-Array zum Zwischenspeichern:

```
    Private fileBytes() As Byte = Nothing
```

QuellDatei in den Puffer einlesen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim strm1 As New FileStream(QuellDatei, FileMode.Open)
        Dim len As Integer = CType(strm1.Length, Integer)
        ReDim fileBytes(len)
        strm1.Read(fileBytes, 0, fileBytes.Length)
        strm1.Close()
    End Sub
```

Datei komprimieren:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim strm2 As New FileStream(KompDatei, FileMode.Create)
        Dim compStrm As New GZipStream(strm2, CompressionMode.Compress)
        compStrm.Write(fileBytes, 0, fileBytes.Length)
        compStrm.Flush()          ' internen Puffer leeren
        compStrm.Close()
        strm2.Close()
    End Sub
```

Datei dekomprimieren:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim strm3 As New FileStream(KompDatei, FileMode.Open)
```

[1] Seit .NET 4.0 eignet sich die *GZipStream* Klasse auch zum Komprimieren von Dateien größer 4 GB!

BEISPIEL 8.20: Mit der *GZipStream*-Klasse wird der Inhalt einer Datei gepackt und entpackt.

```
        Dim decompStrm As New GZipStream(strm3, CompressionMode.Decompress)
        Dim reader As New StreamReader(decompStrm)
        File.WriteAllText(ZielDatei, reader.ReadToEnd())
        reader.Close()
        strm3.Close()
    End Sub
End Class
```

8.7 Memory Mapped Files

Was früher nur mit umständlichen API-Aufrufen möglich war, ist seit .NET 4 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.

[1] IPC = Interprocess Communication

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

VB
```
Imports System.IO.MemoryMappedFiles        ' neu ab.NET 4.0
Imports System.IO
...
Private 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 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.

CreateNew-Methode

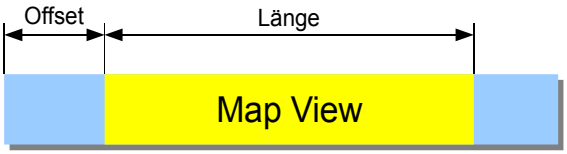
Mit *CreateNew* erzeugen Sie ein MMF das nicht auf eine existierende Datei gemappt ist.

BEISPIEL 8.22: Erzeugen eines MMF mit der maximalen Kapazität von 10000 Bytes

VB
```
Dim cap As Long = 10000
Dim mmf As MemoryMappedFile = 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.

```
VB  Dim offset As Long = &H100000 ' 1 MB
    Dim länge As Long = &HA00000 ' 10 MB
    Dim vacc As MemoryMappedViewAccessor = 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

```
VB  Dim nr As Integer = 12345
    vacc.Write(0, nr)
```

BEISPIEL 8.25: Schreiben eines benutzerdefinierten Typs (generischen Version)

```
VB  vacc.Write(Of Container)(4, MyContainer)
```

BEISPIEL 8.26: Lesezugriff auf das i-te *Byte*

```
VB  vacc.Read(i, 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.

Im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File (MMF) verwenden

wird eine Bilddatei mit Hilfe eines MMF manipuliert.

8.8 Praxisbeispiele

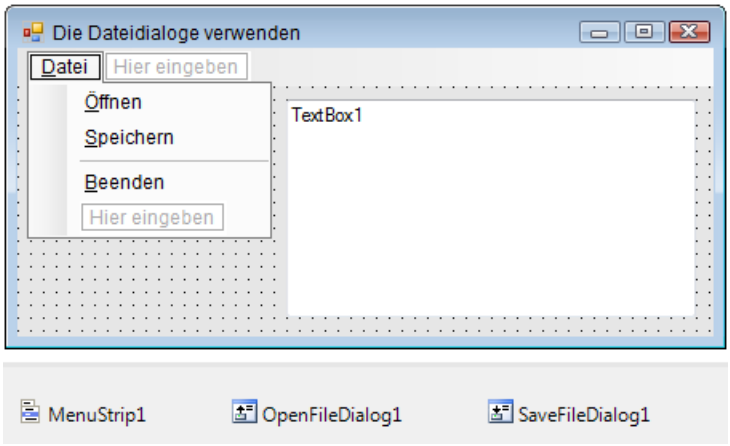
8.8.1 Auf eine Textdatei zugreifen

Das vorliegende Beispiel demonstriert das Öffnen und Speichern einer einfachen Textdatei in Zusammenarbeit mit einer Menükomponente und den Dateidialogen (siehe 8.3.1).

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

```
Imports System.IO

Public Class Form1
```

Der Dateipfad wird zweckmäßig auf Klassenebene deklariert:

```
    Private pfad As String
```

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 Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Zunächst der Öffnen-Dialog, wir beginnen mit dem Festlegen der Standard-Dateinamenserweiterung:

```
        With OpenFileDialog1
            .DefaultExt = "txt"
```

Die Filterzeichenfolge:

```
            .Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*"
```

Warnung, wenn der Namen einer nicht vorhandenen Datei eingegeben wird:

```
            .CheckFileExists = True
```

Das Anfangsverzeichnis:

```
            .InitialDirectory = Application.ExecutablePath
```

Die Beschriftung der Titelleiste des Dialogs:

```
            .Title = "Bitte öffnen Sie eine Textdatei!"
        End With
```

Nun zum Speichern-Dialog:

```
        With SaveFileDialog1
            .DefaultExt = "txt"
```

Der standardmäßig eingetragene Dateiname:

```
            .FileName = "Beispiel.txt"
```

Automatisches Anhängen der *DefaultExt*, falls diese weggelassen wird:

```
            .AddExtension = True
```

Warnung, wenn bereits eine gleichnamige Datei vorhanden ist:

```
            .OverwritePrompt = True
```

Überprüfen, ob Dateiname erlaubte Zeichen enthält:

```
            .ValidateNames = True
```

Weitere Einstellungen:

```
            .Filter = "Textdateien (*.txt)|*.txt|Alle Dateien (*.*)|*.*"
            .InitialDirectory = Application.ExecutablePath
            .Title = "Bitte speichern Sie die Textdatei!"
        End With
    End Sub
```

Den Rahmencode für die *Click*-Eventhandler der einzelnen Menüeinträge erzeugen wir am einfachsten durch Doppelklick auf den entsprechenden Eintrag:

Der Menüpunkt *Datei|Öffnen*:

```
    Private Sub öffnenToolStripMenuItem_Click(sender As Object, e As EventArgs) _
                                              Handles öffnenToolStripMenuItem.Click
        If OpenFileDialog1.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
            pfad = OpenFileDialog1.FileName
            TextBox1.Text = String.Empty
            Try
                TextBox1.Text = File.ReadAllText(pfad)
            Catch
            End Try
```

Dateipfad in der Titelleiste des Formulars anzeigen:

```
            Me.Text = pfad
        End If
    End Sub
```

Der Menüpunkt *Datei|Speichern*:

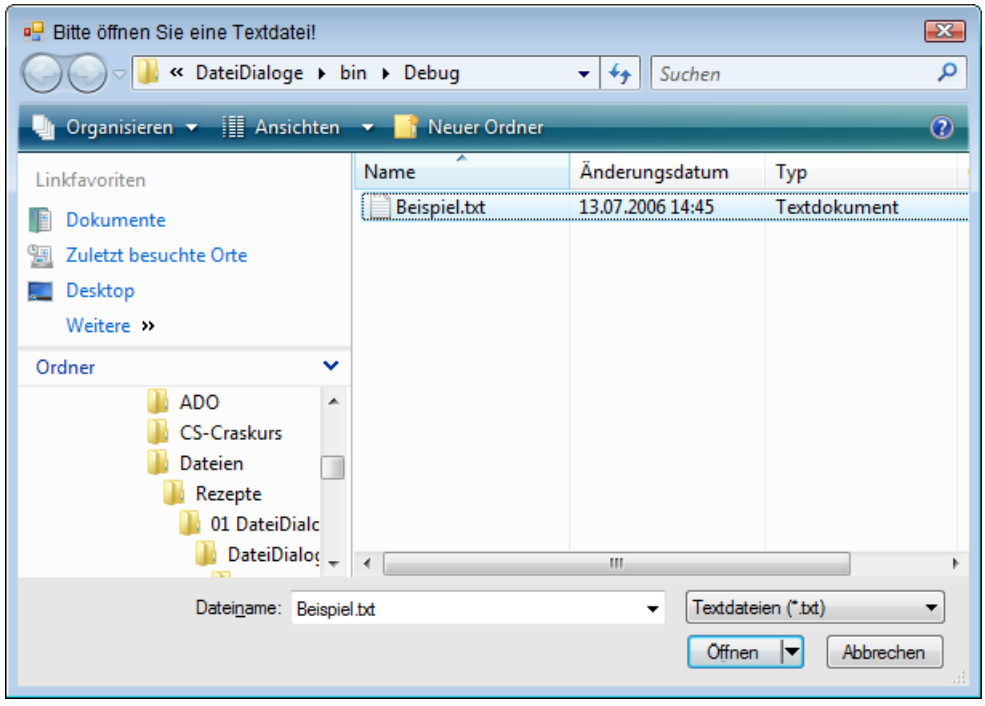
```
    Private Sub speichernToolStripMenuItem_Click(sender As Object, e As EventArgs) _
                                                 Handles speichernToolStripMenuItem.Click
        If SaveFileDialog1.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
            pfad = SaveFileDialog1.FileName
            File.WriteAllText(pfad, TextBox1.Text)
            Me.Text = pfad
        End If
    End Sub
```

Der Menüpunkt *Datei|Beenden*:

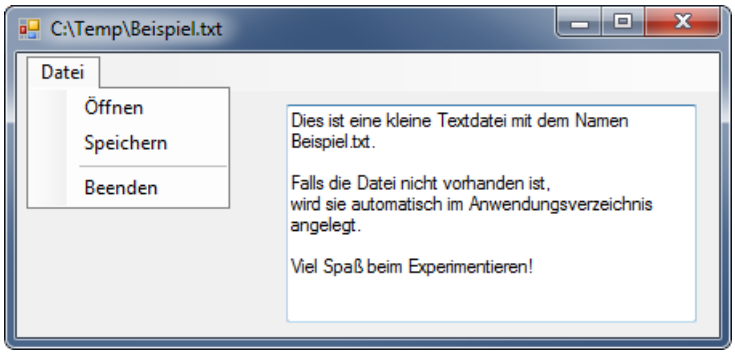
```
    Private Sub beendenToolStripMenuItem_Click(sender As Object, e As EventArgs) _
                                               Handles beendenToolStripMenuItem.Click
        Me.Close()
    End Sub
End Class
```

Test

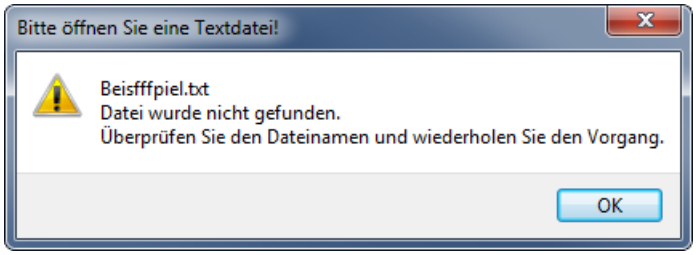
Durch eigene Experimente lässt sich am besten die Wirksamkeit der zahlreichen Eigenschaften der Dateidialoge erkunden:



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 speichern

In diesem Beispiel soll gezeigt werden, wie man auch ohne Datenbank und ADO.NET komplexere Datenstrukturen mittels *BinaryFormatter* auf der Festplatte speichern kann und wie einfach die Benutzerschnittstelle die Datenbindung mittels *BindingSource*- und *BindingNavigator*-Komponenten realisiert.

Klassendiagramm

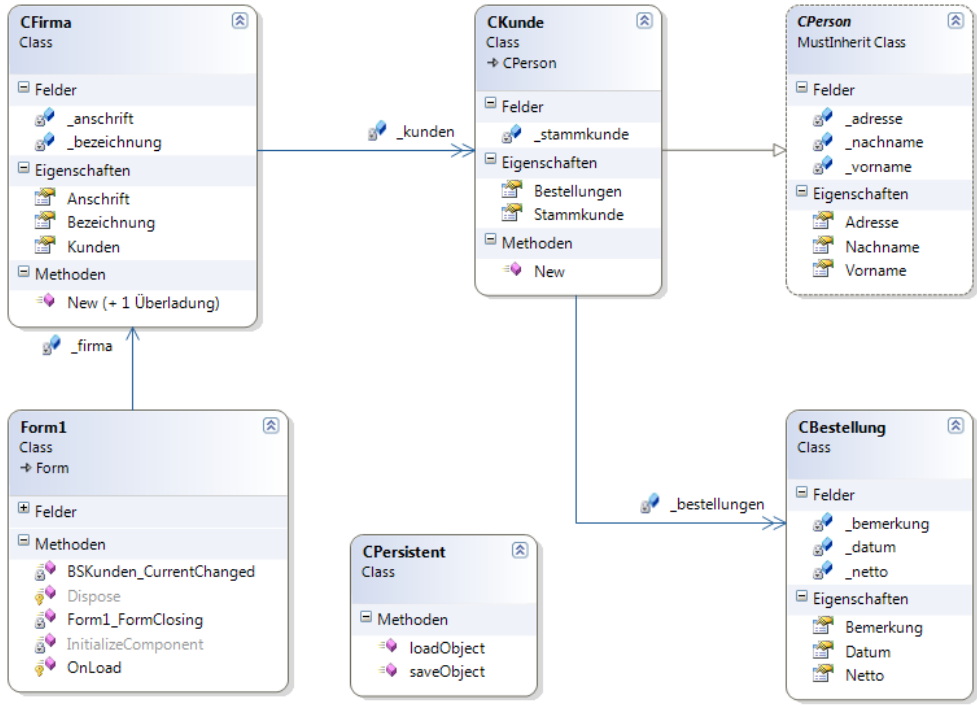
Ausgangspunkt ist das mit dem Klassendesigner von Visual Studio erstellte Klassendiagramm (siehe Kapitel 25), welches neben dem Startformular *Form1* die Klassen *CFirma*, *CKunde*, *CPerson*, *CBestellung* und *CPersistent* enthält.

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).

Eine gleichartige Beziehung besteht auch zwischen den Klassen *CKunde* und *CBestellung* (ein Kunde kann keine, eine oder mehrere Bestellungen haben). Man erkennt aus diesen Zusammen-

[1] Eine Auflistungszuordnung ist – im Unterschied zur einfachen Zuordnung – im Klassen Designer an den doppelten Pfeilspitzen erkennbar.

hängen, 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
    Public Property Datum As DateTime
    Public Property Netto As Decimal
    Public Property Bemerkung As String
End Class
```

Klasse CPerson

Diese abstrakte Klasse stellt ihren Nachkommen vier allgemeine Eigenschaften als "Erbmaterial" zur Verfügung:

```
<Serializable()> Public MustInherit Class CPerson
```

Selbst implementierende Eigenschaften sparen Schreibarbeit:

```
   Public Property Vorname As String
   Public Property Nachname As String
   Public Property Adresse() As String
End Class
```

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
    Inherits CPerson
```

Eine generische Collection speichert die Bestellungen eines Kunden:

```
      Private _bestellungen As IList(Of CBestellung)
```

Im Konstruktor wird die (zunächst leere) Liste der Bestellungen erzeugt:

```
    Sub New()
        _bestellungen = New List(Of CBestellung)
    End Sub
```

Der Lese- und Schreibzugriff:

```
    Public Property Bestellungen() As IList(Of CBestellung)
        Get
            Return _bestellungen
        End Get
        Set(value As IList(Of CBestellung))
            _bestellungen = value
        End Set
    End Property
```

Eine selbst implementierende Eigenschaft:

```
   Public Property Stammkunde() As Boolean
End Class
```

Klasse CFirma

Diese Klasse ist die Wurzelklasse des Objektbaums und kapselt neben allgemeinen Eigenschaften (*Bezeichnung* und *Anschrift*) lediglich die Kundenliste (generische Liste vom Typ *CKunde*).

```
<Serializable()> Public Class CFirma
```

Eine generische Collection speichert die Kunden der Firma:

```
    Private _kunden As IList(Of CKunde)
```

Im Konstruktor wird die (zunächst leere) Liste der Kunden erzeugt:

```
    Sub New()
       _kunden = New List(Of CKunde)
    End Sub
```

Der Zugriff auf die Kundenliste:

```
    Public Property Kunden As IList(Of CKunde)
        Get
            Return _kunden
        End Get
        Set(value As IList(Of CKunde))
            _kunden = value
        End Set
    End Property
```

Zwei selbst implementierende Eigenschaften:

```
    Public Property Bezeichnung As String
    Public Property Anschrift As String
End Class
```

Klasse CPersistent

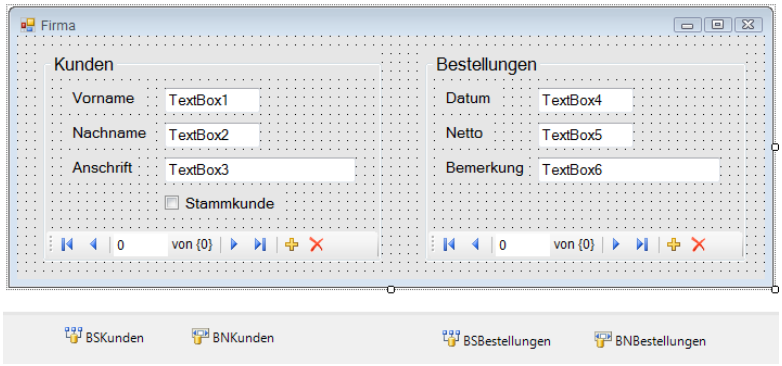
Diese Klasse exportiert die statischen Methoden *saveObject* und *loadObject*, mit denen die Serialisierung/-Deserialisierung beliebiger Objekte möglich ist. Die Methode *saveObject* übernimmt als Parameter das Objekt und den Dateipfad, serialisiert das Objekt und speichert es auf der Festplatte ab. Die Methode *loadObject* erwartet als Parameter den Dateipfad, holt sich das Objekt von der Festplatte und liefert es deserialisiert zurück.

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Public Class CPersistent
    Public Shared Sub saveObject(o As Object, pfad As String)
        Dim fs As New FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None)
        Dim bf As New BinaryFormatter()
        bf.Serialize(fs, o)
        fs.Close()
    End Sub

    Public Shared Function loadObject(pfad As String) As Object
        Dim fs As New FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read)
        Dim bf As New BinaryFormatter()
        Dim o As Object = bf.Deserialize(fs)
```

```
        fs.Close()
        Return o
    End Function
End Class
```

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 Class Form1
```

Man sieht es dieser Variablen nicht an, dass sie den kompletten Objektbaum kapselt:

```
    Private _firma As New CFirma()
```

Der Dateipfad verweist in unserem Fall auf das Ausgabeverzeichnis des Projekts:

```
    Private Const PFAD As String = "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 Overrides Sub OnLoad(e As EventArgs)
        Try
            _firma = CType(CPersistent.loadObject(PFAD), CFirma)
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
```

Die erste *BindingSource* wird an die Kundenliste gebunden:

```
        BSKunden.DataSource = _firma.Kunden
```

Die Steuerelemente an die *BindingSource* koppeln:

```
        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))
        MyBase.OnLoad(e)
    End Sub
```

Wenn zu einem anderen Kunden gewechselt wird, müssen auch die zu diesem Kunden gehörenden Bestellungen ermittelt und der *BindingSource* zugewiesen werden:

```
    Private Sub BSKunden_CurrentChanged(sender As Object, e As EventArgs) _
                                                    Handles BSKunden.CurrentChanged
```

Der Kunde wird ermittelt:

```
        Dim kunde As CKunde = CType(BSKunden.Current, CKunde)
```

Die zweite *BindingSource* wird an die Liste der Bestellungen gebunden:

```
        BSBestellungen.DataSource = kunde.Bestellungen
```

Die Steuerelemente müssen 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)
    End Sub
```

Beim Schließen des Formulars wird der komplette Objektbaum abgespeichert:

```
    Private Sub Form1_FormClosing(sender As Object,
                                  e As FormClosingEventArgs) Handles Me.FormClosing
        Try
            CPersistent.saveObject(_firma, PFAD)
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
    End Sub
End Class
```

Seit VB 2015 gibt es mit der *NameOf*-Funktion einen typsicheren Weg zur Einbettung von Stringliteralen in den Code, zum Beispiel könnte jetzt anstatt

```
TextBox1.DataBindings.Add(New Binding("Text", BSKunden, "Vorname", True))
```

geschrieben werden:

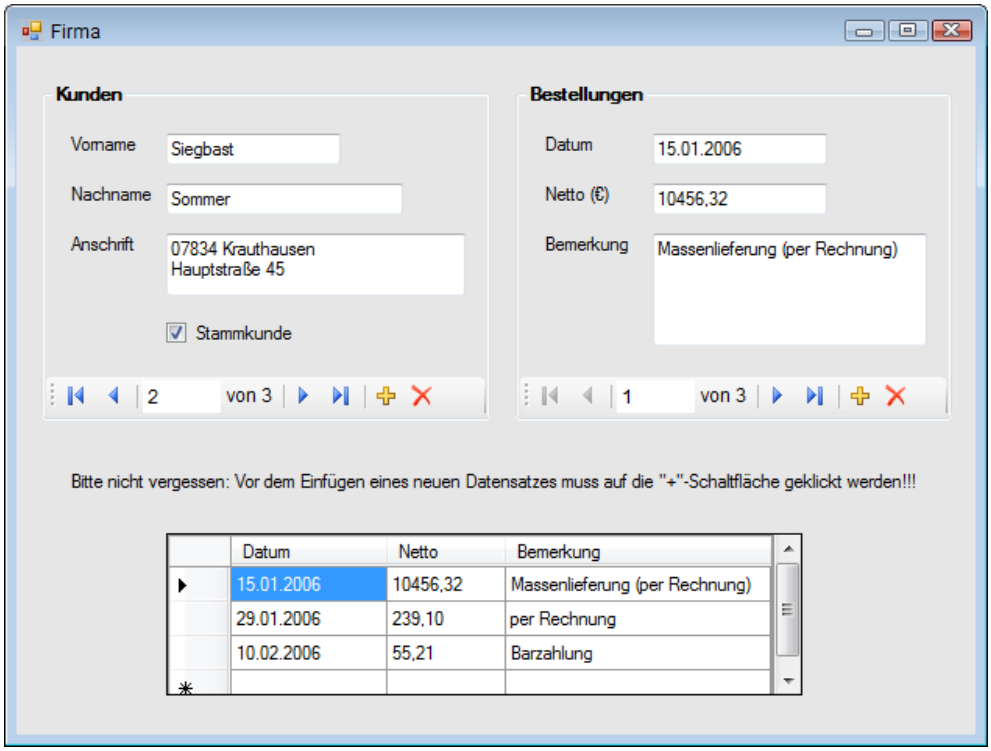
```
TextBox1.DataBindings.Add(New Binding("Text", BSKunden, NameOf(kunde.Vorname), True))
```

In diesem Fall müsste allerdings die Variable *kunde* global deklariert werden.

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 man auch die Bedeutung der *Using*-Klausel kennen.

Oberfläche

Auf das Startformular einer Windows Forms-Anwendung setzen Sie einen *Button* und eine *PictureBox*.

Quellcode

```
Imports System.IO
Imports System.IO.MemoryMappedFiles
Imports System.Runtime.InteropServices

Public Class Form1
    ...
```

Eine RGB-Farbstruktur definieren (inkl. einer Methode):

```
    Public Structure MyRGB
        Public Red As Short
        Public Green As Short
        Public Blue As Short
        Public Alpha As Short
```

Jetzt werden die Farbanteile verfälscht (der Einsatz der mathematischen *Min*-Funktion verhindert hier eine Bereichsüberschreitung):

```
        Public Sub changeRGB(value As Short)
            Red = CShort(Fix(Math.Min(Short.MaxValue, CInt(Fix(Red)) + value)))
            Green = CShort(Fix(Math.Min(Short.MaxValue, CInt(Fix(Green)) + value)))
            Blue = CShort(Fix(Math.Min(Short.MaxValue, CInt(Fix(Blue)) + value)))
            Alpha = CShort(Fix(Math.Min(Short.MaxValue, CInt(Fix(Alpha)) + value)))
        End Sub
    End Structure
```

Die Start-Schaltfläche:

```
   Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

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 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:

```
            Dim offset As Long = &H100000   ' 1 MB
            Dim länge As Long = &HA00000    ' 10 MB

            Using vacc = mmf.CreateViewAccessor(offset, länge)
```

Anzahl der Bytes pro Farbwert ermitteln:

```
                Dim colorSize As Integer = Marshal.SizeOf(GetType(MyRGB))
```

Eine Puffervariable für den einzulesenden Farbwert:

```
                Dim color As MyRGB
```

Den Map View manipulieren:

```
                For i As Long = 0 To länge - 1 Step colorSize
```

Vom MMF in *color*-Variable einlesen

```
                    vacc.Read(i, color)
```

Die *color*-Variable manipulieren:

```
                    color.changeRGB(20)
```

Die *color*-Variable zurück ins MMF schreiben:

```
                    vacc.Write(i, color)
                Next i
            End Using
        End Using
```

Bild anzeigen:

```
        PictureBox1.Load("Bild1.bmp")
    End Sub
End Class
```

Test

Starten Sie das Programm und klicken Sie dann mehrfach hintereinander auf die Schaltfläche. Beobachten Sie, wie sich die Farbwerte des Bildes verändern.

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[1]).
- Einen nützlichen Rechner zur Dezimal-Hexadezimal-MByte-Konvertierung finden Sie im folgenden Praxisbeispiel.

8.8.4 Hex-Dezimal-Bytes-Konverter

Mit Umrechnungen zwischen Hex- und Dezimalzahlen und Bytegrößen wird der Programmierer ziemlich häufig gequält (z.B. bei Arbeit mit Pointern oder MMF[2]s).

Zuweisen eines Speicherbereichsabschnitts zwischen 256 MB und 768 MB:

```
Dim offset As Long = 0x10000000   ' 256 MB
Dim length As Long = 0x20000000   ' 512 MB
```

Die Endadresse von 768 Megabyte ergibt sich aus der Summe von *offset* und *length*.

Die Adressangaben im obigen Beispiel sind Hexadezimalzahlen.Wie man sieht, lassen sich Hexzahlen den *Long*- bzw. *Int64*-Werten direkt zuweisen, indem man den Präfix *0x* voranstellt. Doch wie kommt man von einer Byte-Adressangabe zu der äquivalenten Hexzahl? Und wie ist der umgekehrte Weg? Bei der Umrechnung lauert die erste Falle:

HINWEIS: Die Hexzahl gibt die Bytes an (nicht Bits!)

Schluss mit weiteren zeit- und nervtötenden Rechnereien, wir werden einen praktikablen "Taschenrechner" für diese Zwecke entwickeln, wollen jedoch vorher einige grundlegenden Beziehungen klären.

Hex-Dezimal-Konvertierungen

Zunächst sollten wir uns mit der Hex-Dezimal-Umrechnung in beiden Richtungen beschäftigen. Dazu eignen sich die folgenden beiden Funktionen:

```
Public Function HexToDec(hexValue As String) As Integer        ' Hexadezimal nach Dezimal
    Return Int32.Parse(hexValue, System.Globalization.NumberStyles.HexNumber)
End Function

Public Function DecToHex(decValue As Long) As String          ' Dezimal nach Hexadezimal
   Return String.Format("{0:x}", decValue)
End Function
```

Verwendung obiger Funktionen:

```
Dim hex As String = DecToHex(255)       ' liefert die Hexzahl FF
```

---

[1] ... und der nimmt sich manchmal recht viel Zeit.

[2] Memory Mapped File

```
Dim dec As Integer = HexToDec("FF")     ' liefert die Integer-Zahl 255
```

Byte-Kilobyte-Megabyte-Gigabyte-Konvertierungen

Es gelten die folgenden Umrechnungen:

```
KB = Byte / 1024
MB = KB / 1024
GB = MB / 1024
```

Umrechnung Byte in Megabyte:

```
MB = Byte / (1024 * 1024)
```

oder

```
MB = Byte / 1048576
```

Oberfläche

Auf ein Windows-Formular setzen wir fünf Textboxen mit entsprechenden Labels zur Beschriftung (siehe Laufzeitabbildung).

Quellcode

```
Public Class Form1
```

Für jede der umzurechnenden Größen deklarieren wir eine Zustandsvariable geeigneten Datentyps:

```
    Private kbytes, mbytes, gbytes As Single
    Private dez As Long
    Private hex As String
```

Da wir auf eine Ergebnistaste verzichtet haben, müssen der Umrechnungsvorgang und die Aktualisierung der Anzeige der anderen Textboxen automatisch bei jedem Loslassen einer Taste gestartet werden. Dazu benutzen wir das *KeyUp*-Event jeder einzelnen *TextBox*. Weil sich nicht jede Texteingabe in den gewünschten Typ umwandeln lässt und auch leere Textfelder zu einem Programmabbruch führen würden, werden die kritischen Konvertierungen mit der statischen Methode *TryParse* (anstatt mit *Parse*) vorgenommen.

Die Eingabe einer Dezimalzahl (hier auch identisch mit der Eingabe der Anzahl Bytes):

```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If Int64.TryParse(TextBox1.Text, dez) Then
```

Die Umrechnung:

```
            hex = String.Format("{0:x}", dez)
            kbytes = Convert.ToSingle(dez) / 1024
            mbytes = kbytes / 1024
            gbytes = mbytes / 1024
```

Die Anzeige:

```
            TextBox2.Text = hex.ToUpper()
            TextBox3.Text = kbytes.ToString("#,##0.000")
            TextBox4.Text = mbytes.ToString("#,##0.000")
            TextBox5.Text = gbytes.ToString("#,##0.000")
        End If
    End Sub
```

Die Eingabe einer Hexadezimalzahl:

```
    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        If Int64.TryParse(TextBox2.Text, System.Globalization.NumberStyles.HexNumber,
                                                              Nothing, dez) Then
            kbytes = Convert.ToSingle(dez) / 1024
            mbytes = kbytes / 1024
            gbytes = mbytes / 1024
            TextBox1.Text = dez.ToString()
            TextBox3.Text = kbytes.ToString("#,##0.000")
            TextBox4.Text = mbytes.ToString("#,##0.000")
            TextBox5.Text = gbytes.ToString("#,##0.000")
        End If
    End Sub
```

Die KByte-Eingabe:

```
    Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        If Single.TryParse(TextBox3.Text, kbytes) Then
            dez = Convert.ToInt64(kbytes) * 1024
            hex = String.Format("{0:x}", dez)
            mbytes = kbytes / 1024
            gbytes = mbytes / 1024
            TextBox1.Text = dez.ToString
            TextBox2.Text = hex.ToUpper
            TextBox4.Text = mbytes.ToString("#,##0.000")
            TextBox5.Text = gbytes.ToString("#,##0.000")
        End If
    End Sub
```

Die MByte-Eingabe:

```
    Private Sub TextBox4_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox4.KeyUp
        If Single.TryParse(TextBox4.Text, mbytes) Then
            kbytes = mbytes * 1024
            dez = Convert.ToInt64(kbytes) * 1024
            hex = String.Format("{0:x}", dez)
            mbytes = kbytes / 1024
            gbytes = mbytes / 1024
            TextBox1.Text = dez.ToString
            TextBox2.Text = hex.ToUpper
            TextBox3.Text = kbytes.ToString("#,##0.000")
            TextBox5.Text = gbytes.ToString("#,##0.000")
```

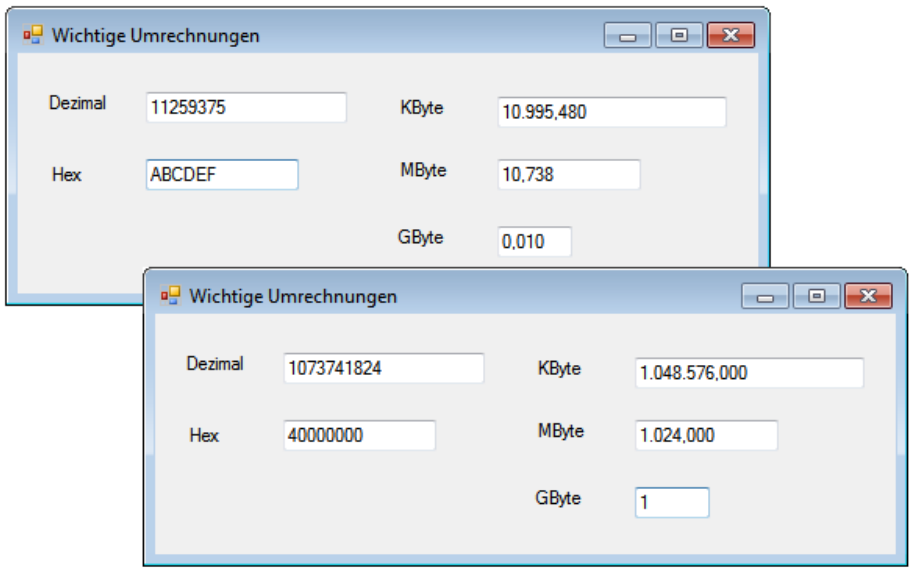
```
        End If
    End Sub
```

Und schließlich die GByte-Eingabe:

```
    Private Sub TextBox5_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox5.KeyUp
        If Single.TryParse(TextBox5.Text, gbytes) Then
            mbytes = gbytes * 1024
            kbytes = mbytes * 1024
            dez = Convert.ToInt64(kbytes) * 1024
            hex = String.Format("{0:x}", dez)
            TextBox1.Text = dez.ToString
            TextBox2.Text = hex.ToUpper
            TextBox3.Text = kbytes.ToString("#,##0.000")
            TextBox4.Text = mbytes.ToString("#,##0.000")
        End If
    End Sub
End Class
```

Test

Sie können in ein beliebiges Feld eine Zahl eingeben, wie von Geisterhand werden die anderen Felder sofort nach jedem Zeichen aktualisiert.



Bemerkungen

- Die Hexzahlen können Sie in Groß- oder Kleinbuchstaben eingeben. Ausgegeben werden aber nur Großbuchstaben.
- Für die Byte- Eingabe benutzen Sie das *Dezimal*-Feld. Es sind nur Ganzzahlen zulässig!

- Wie Sie sehen, haben wir für die Anzeige von KByte, MByte und GByte je drei Nachkommastellen vorgesehen. Das können Sie durch Anpassen des Formatierungsstrings ändern.
- Verwechseln Sie nicht Dezimalpunkt mit Dezimalkomma (der Unterschied ist bei manchen Schrifttypen und -größen nur schwer zu erkennen).

8.8.5 Eine Datei verschlüsseln

Dieses Beispiel demonstriert das Erzeugen eines *CryptoStreams* zum symmetrischen Verschlüsseln bzw. Entschlüsseln einer Datei nach dem *Data Encryption Standard* (DES)[1].

Oberfläche

Auf das Startformular *Form1* setzen Sie zwei Schaltflächen und zwei *TextBox*en (siehe Laufzeitansicht).

Quellcode

```
Imports System.Text
Imports System.IO
Imports System.Security.Cryptography

Public Class Form1
```

Auf globaler Ebene wird zunächst die Instanz einer DES Implementierung erzeugt. Da wir dem Konstruktor keine Argumente übergeben, wird ein Zufallsschlüssel generiert und die Standardeigenschaften entsprechen den üblichen Verschlüsselungs-Szenarien:

```
    private des As New DESCryptoServiceProvider()
```

Der Name der Datei, die angelegt und verschlüsselt werden soll:

```
    Private fileName As String = "EncryptedFile.txt"
```

Eine Hilfsroutine, welche einen String in ein Byte-Array transformiert:

```
    Public Function ConvertStringToByteArray(s As String) As Byte()
        Dim ucenc As New UnicodeEncoding()
        Return (ucenc.GetBytes(s))
    End Function
```

Die Schaltfläche *Verschlüsseln*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Den zu verschlüsselnden Text in den Byte-Array-Puffer kopieren:

```
        Dim byteArr() As Byte = ConvertStringToByteArray(TextBox1.Text)
```

[1] Gemeint ist also nicht die einfache Verschlüsselung von Verzeichnissen und Dateien, wie sie z B. mittels *Encrypt-/Decrypt*-Methoden der *File*-Klasse möglich ist.

Ein DES Encryptor Objekt wird auf einer DES Instanz erzeugt:

```
            Dim desEncrypt As ICryptoTransform = des.CreateEncryptor()
```

Der *FileStream*, welcher die verschlüsselte Datei schreiben soll, wird erzeugt:

```
            Dim fs As New FileStream(fileName, FileMode.Create, FileAccess.Write)
```

Der *CryptoStream,* der den *FileStream* verschlüsseln soll*,* wird erzeugt, wobei der Konstruktor eine *FileStream*-Instanz und den DES Encryptor erhält. Der Stream wird in den Write-Modus versetzt:

```
            Dim cryptoStrm As New CryptoStream(fs, desEncrypt, CryptoStreamMode.Write)
```

Schließlich schreiben wir das mit unserem Text gefüllte Byte-Array in den Stream und schließen diesen. Als Resultat entsteht die verschlüsselte Datei *EncryptedFile.txt*.

```
            cryptoStrm.Write(byteArr, 0, byteArr.Length)
            cryptoStrm.Close()
        End Sub
```

Die Schaltfläche *Entschlüsseln*:

```
        Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Den Decryptor der vorhandenen DES-Instanz erzeugen:

```
            Dim desDecrypt As ICryptoTransform = des.CreateDecryptor()
```

Den Filestream erzeugen, um die verschlüsselte Datei einzulesen:

```
            Dim fs As New FileStream(fileName, FileMode.Open, FileAccess.Read)
```

Den Kryptostream erzeugen, um die verschlüsselten Bytes zu entschlüsseln:

```
          Try
```

Kryptostream erzeugen, um die verschlüsselten Bytes zu entschlüsseln:

```
            Dim cryptoStrm As New CryptoStream(fs, desDecrypt, CryptoStreamMode.Read)
```

Inhalt auslesen und entschlüsseln:

```
            Dim strmRead As New StreamReader(cryptoStrm, New UnicodeEncoding())
```

Inhalt anzeigen:

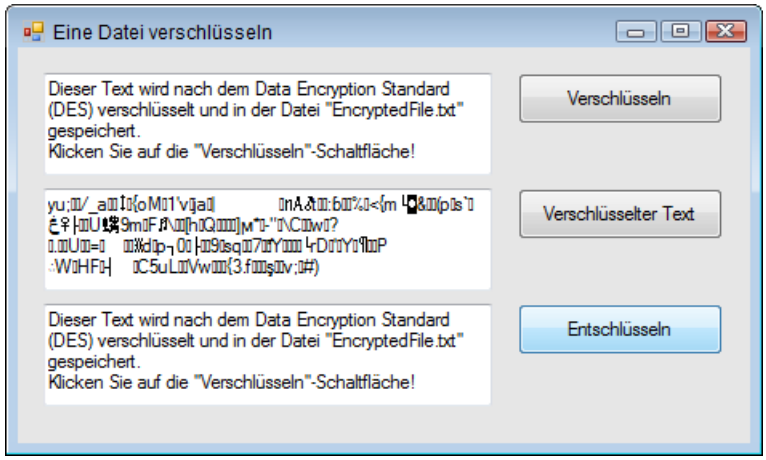
```
            TextBox2.Text = strmRead.ReadToEnd()
            cryptoStrm.Close()
        Catch ex As CryptographicException
            MessageBox.Show(ex.Message, "Falscher Schlüssel")
        Finally
            fs.Close()
        End Try
    End Sub
```

Praktisch ziemlich nutzlos, aber trotzdem interessant, ist ein Blick auf den Inhalt der verschlüsselten Datei:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        TextBox3.Text = File.ReadAllText(fileName)
    End Sub
End Class
```

Test

Geben Sie einen beliebigen Text in das obere Textfeld ein und klicken Sie die *Verschlüsseln*-Schaltfläche. Den Inhalt der verschlüsselten Datei *EncryptedFile.txt* können Sie sich nach Klick auf die mittlere Schaltfläche betrachten.



Über die *Entschlüsseln*-Schaltfläche erhalten Sie wieder den ursprünglichen Test.

HINWEIS: Es ist praktisch unmöglich, ohne Kenntnis des Schlüssels aus dem verschlüsselten Text wieder das Original zu rekonstruieren.

Da nach einem erneuten Programmstart auch ein neuer Schlüssel angelegt wird, führt der sofortige Klick auf die *Entschlüsseln*-Schaltfläche zu einer Fehlermeldung.

Bemerkung

Falls die verschlüsselte Datei weitergegeben werden soll, benötigt der Empfänger natürlich exakt den Schlüssel, mit dem die Datei verschlüsselt wurde. Dieser entspricht der *Key*-Eigenschaft (ein Byte-Array) des *DESCryptoServiceProvider*-Objekts:

```
File.WriteAllBytes("Key.dat", des.Key)        ' Absender schreibt Schlüsseldatei
des.Key = File.ReadAllBytes("Key.dat")        ' Empfänger liest Schlüsseldatei
```

8.8.6 Eine Datei komprimieren

In diesem Beispiel wollen wir zeigen, wie unter Verwendung der *GZipStream*-Klasse der Inhalt einer Datei gepackt und entpackt werden kann.

HINWEIS: Die *GZipStream*-Klasse eignet sich vor .NET 4 nur zum Komprimieren von Dateien kleiner vier GB.

Oberfläche

Auf das Startformular *Form1* setzen wir Schaltflächen mit den Beschriftungen "Quelldatei lesen", "Komprimieren" und "Dekomprimieren". Damit dürften auch die drei grundlegenden Operationen klar sein, die wir mit unserem Testprogramm durchführen wollen.

Quellcode

```
Imports System.IO
Imports System.IO.Compression

Public Class Form1
```

Die Namen der drei Dateien, die sich alle im Anwendungsverzeichnis befinden:

```
    Private QuellDatei As String = "Test1.txt"
    Private KompDatei As String = "Test.zip"
    Private ZielDatei As String = "Test2.txt"
```

Ein Byte-Puffer:

```
    Private fileBytes() As Byte = Nothing
```

Die Quelldatei in den Puffer einlesen:

```
        Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim strm1 As New FileStream(QuellDatei, FileMode.Open)
        Dim lg As Integer = Convert.ToInt32(strm1.Length)
        ReDim fileBytes(lg)
        strm1.Read(fileBytes, 0, fileBytes.Length)
        strm1.Close()
    End Sub
```

Die komprimierte Datei erstellen:

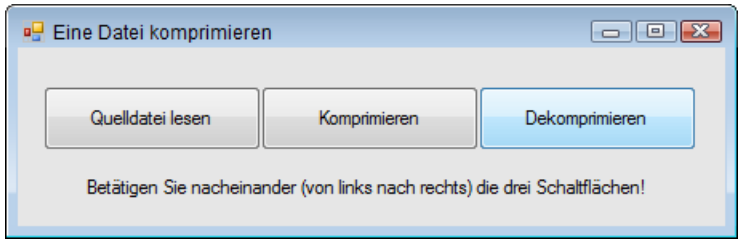
```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim strm2 As New FileStream(KompDatei, FileMode.Create)
        Dim compStrm As New GZipStream(strm2, CompressionMode.Compress)
        compStrm.Write(fileBytes, 0, fileBytes.Length)
        compStrm.Flush()    ' internen Puffer leeren
        compStrm.Close()
        strm2.Close()
    End Sub
```

Die Datei dekomprimieren:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim strm3 As New FileStream(KompDatei, FileMode.Open)
        Dim decompStrm As New GZipStream(strm3, CompressionMode.Decompress)
        Dim reader As New StreamReader(CType(decompStrm, Stream))
        File.WriteAllText(ZielDatei, reader.ReadToEnd())
        reader.Close()
        strm3.Close()
    End Sub
End Class
```

Test

Speichern Sie zunächst eine beliebige Textdatei unter dem Namen *Test1.txt* im *\bin\Debug*-Unterverzeichnis des Projektordners ab. Nach dem Programmstart betätigen Sie nacheinander (von oben nach unten) die drei Schaltflächen.



Im Ergebnis finden sich im Anwendungsverzeichnis neben der Quell-Datei *Test1.txt* die beiden neu erzeugten Dateien *Test.zip* und *Test2.txt*. Vergleichen Sie die Inhalte von *Test1.txt* und *Test2.txt*, so müssen diese identisch sein.

HINWEIS: Falls es sich, wie in unserem Beispiel, um sehr kleine Dateien handelt (siehe folgende Abbildung) ist ein Kompressionseffekt allerdings kaum nachweisbar.

Name	Änderungsdatum	Typ	Größe
Compress.exe	13.03.2008 16:39	Anwendung	9 KB
Compress.pdb	13.03.2008 16:39	Program Debug Database	34 KB
Compress.vshost.exe	13.03.2008 16:38	Anwendung	10 KB
Compress.vshost.exe....	21.07.2007 02:33	MANIFEST-Datei	1 KB
Test1.txt	15.03.2006 16:19	Textdokument	1 KB
Test2.txt	13.03.2008 16:39	Textdokument	1 KB
Test.zip	13.03.2008 16:39	ZIP-komprimierter Ordner	1 KB

8.8.7 Echte ZIP-Dateien erstellen

Geht es um das Erstellen und Verarbeiten von ZIP-Dateien, können Sie sicher beim "Urschleim" anfangen und eine eigene Library erstellen. Sie können sich aber auch das Leben leicht machen und einfach eine der vielen Freeware-Libraries nutzen, die das Web bietet.

Wir haben uns für die "DotNetZip Library" entschieden, die einen recht großen Funktionsumfang bietet, gleichzeitig aber auch einfach und intuitiv programmierbar ist. Die Library und die dazugehörige Hilfedatei finden Sie unter:

LINK: `http://dotnetzip.codeplex.com/`

Entpacken Sie die heruntergeladene Datei und binden Sie die enthaltene Datei *Ionic.Zip.dll* als Verweis in Ihre Anwendung ein. Die Option *Copy Local* setzen Sie bitte auf *True*, in diesem Fall wird die DLL in das spätere Anwendungsverzeichnis kopiert.

Oberfläche

Erstellen Sie eine einfache WPF-Anwendung mit einer *ListBox* und einer Schaltfläche:

```
<Window x:Class="ZIP_Bsp.MainWindow"
...
        Title="MainWindow" Height="350" Width="525" Loaded="Window_Loaded">
    <DockPanel >
        <Button DockPanel.Dock="Top" Content="Packen" Height="23" Name="Button1"
                Click="Button1_Click" />
        <ListBox  Name="ListBox1"  />
    </DockPanel>
</Window>
```

Quelltext

```
...
Imports System.IO
Imports Ionic.Zip

Class MainWindow
...
```

Mit dem Laden des Formulars füllen wir die *ListBox* mit den Namen der Datei im aktuellen Verzeichnis:

```
    Private Sub Window_Loaded(sender As Object, e As System.Windows.RoutedEventArgs)
        Dim di As New DirectoryInfo(AppDomain.CurrentDomain.BaseDirectory)
        ListBox1.Items.Clear()
        For Each f As FileInfo In di.GetFiles()
            ListBox1.Items.Add(f.FullName)
        Next f
    End Sub
```

So, nun können wir uns schon dem Verpacken der Dateien widmen:

```
Private Sub Button1_Click(sender As Object, e As System.Windows.RoutedEventArgs)
```

Eine Instanz der *ZipFile*-Klasse erstellen:

```
    Using zip As New ZipFile()
```

Das optionale Passwort festlegen:

```
        zip.Password = "geheim"
```

Alle Dateien, die in der *ListBox* angezeigt werden, der ZIP-Datei hinzufügen (wir legen diese im Wurzelverzeichnis der ZIP-Datei ab, optional können Sie auch die Laufwerksstruktur übernehmen oder einen anderen Pfad innerhalb der ZIP-Datei festlegen:

```
        For Each s As String In ListBox1.Items
            zip.AddFile(s, "\")
        Next s
```

Speichern der ZIP-Datei:

```
        zip.Save(AppDomain.CurrentDomain.BaseDirectory & "Backup.zip")
```

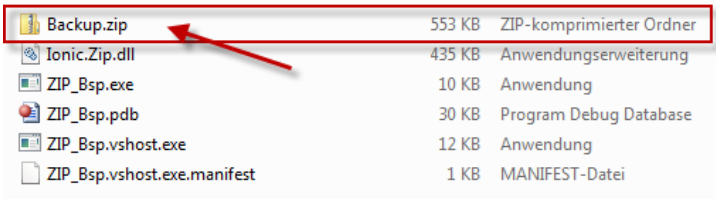
Anzeige des Folders:

```
        Process.Start(AppDomain.CurrentDomain.BaseDirectory)
    End Using
End Sub
```

Ups, das war es schon, weniger Code ist wohl kaum möglich.

Test

Nach dem Start genügt ein kurzer Klick auf die Schaltfläche und Sie erhalten die gewünschte ZIP-Datei:



8.8.8 PDFs erstellen/exportieren

Viele Weg führen nach Rom ... und so finden sich in VB auch reichlich Möglichkeiten, Programmausgaben im bekannten PDF-Format zu Papier zubringen:

- Crystal Report (Exportfunktion)
- Report Services (Exportfunktion)
- diverse Freeware und Shareware-Libraries

Auf die ersten beiden Varianten wollen wir im Rahmen dieses Abschnitts nicht eingehen, nur soviel: Arbeiten Sie mit Datenbankausgaben und einem der beiden o.g. Tools, nutzen Sie am besten deren PDF-Exportfunktionen.

Konnten wir Sie mit den beiden Reportgeneratoren nicht zufrieden stellen, haben Sie vermutlich andere Anforderungen an die Seitengestaltung oder Sie vermissen grundlegende Gestaltungsmöglichkeiten und Optionen der PDF-Ausgabe.

Für all diese Aufgaben bieten sich jedoch die diversen Freeware/Opensource-Lösungen aus dem Web an.

HINWEIS: Eine Wertung der einzelnen Lösungen wollen wir an dieser Stelle nicht vornehmen, wichtig ist uns vor allem, dass die Projekte aktuell noch gepflegt werden und nicht mit einem Versionsstand von 2003 o.ä. "versandet" sind.

Aus den genannten Gründen blieben für uns dann nur die beiden im Folgenden vorgestellten Lösungen übrig.

iTextSharp

Bei dieser Library (ja, es handelt sich um nur eine Datei) handelt es sich um eine Portierung der iText-Java Library mit C# für die .NET-Plattform. Die Assembly bzw. die Sourcecodes können Sie unter folgender Adresse herunterladen:

LINK: `http://sourceforge.net/projects/itextsharp/`

Zum Installieren genügt es, wenn Sie eine Referenz auf die Assembly *itextsharp.dll* erstellen. Ein kleines Beispiel soll Ihnen die prinzipielle Verwendung demonstrieren:

Ein kleines Beispiel für die Verwendung der Library:

```
Imports System.IO
Imports iTextSharp
Imports iTextSharp.text
Imports iTextSharp.text.pdf
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Dokument erstellen:

```
        Dim document As New Document()
```

Die eigentliche Ausgabe festlegen (auch ein *MemoryStream* ist denkbar[1]):

```
        PdfWriter.GetInstance(document, New FileStream("Test.pdf", FileMode.Create))
```

Dokument öffnen und die Dokumenteigenschaften festlegen:

```
        document.Open()
        document.AddTitle("PDF-Experimente")
```

[1] z B. für ASP.NET-Anwendungen, die Ausgaben per PDF zurückgeben wollen

```
        document.AddAuthor("Meyer Müller")
        document.AddSubject("Mein erster Test")
```

Einige Zeilen ausgeben:

```
        document.Add(New Paragraph("Hello World!"))
        document.Add(New Paragraph("Hallo Leser!"))
```

Eine neue Seite anlegen:

```
        document.NewPage()
        document.Add(New Paragraph("Neue Seite"))
```

Das Dokument schließen:

```
        document.Close()
        Process.Start("Test.pdf")
    End Sub
End Class
```

Weitere praktische Beispiele finden Sie unter der folgenden Adresse:

LINK: `http://www.codeproject.com/KB/graphics/iTextSharpTutorial.aspx`

Eine mehr oder weniger fehlerhafte Dokumentation finden Sie unter

LINK: `http://itextsharp.sourceforge.net/tutorial/index.html`

Da die Ausgabe auch absatzweise erfolgen kann (siehe oben), dürfte diese Library für Fließtexte etc. recht gut geeignet sein.

PDFsharp

Einen etwas anderes Konzept als die iTextSharp-Lösung verfolgt PDFSharp. Hier finden Sie sich als alter GDI+-Programmierer recht schnell zurecht, wird doch ein dem *Graphics*-Objekt ähnliches Konzept verfolgt, um Seiten per GDI-Befehlen zusammenzustellen.

Doch zunächst der Link auf den Download:

LINK: `http://www.pdfsharp.net/Default.aspx?AspxAutoDetectCookieSupport=1`

Auch wenn es in diesem Fall mehr als eine Assembly ist, geht die Installation per Referenz einfügen genauso schnell von der Hand. Folgende Dateien sind für eine Windows Foms-Anwendung zu installieren:

Auch hier soll ein kleines Beispiel die Verwendung demonstrieren.

Ein einfaches PDF-Dokument erstellen:

```
...
Imports System.IO
Imports System.Diagnostics
Imports PdfSharp
Imports PdfSharp.Drawing
Imports PdfSharp.Pdf
Imports PdfSharp.Pdf.IO
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Dokument erstellen:

```
        Dim document As New PdfDocument()
```

Einige Dokument-Eigenschaften festlegen:

```
        document.Info.Title = "PDF-Experimente"
        document.Info.CreationDate = Date.Now
        document.Info.Author = "Meyer Müller"
        document.Info.Subject = "Mein erster Test"
```

Die erste Seite einfügen:

```
        Dim page As PdfPage = document.AddPage()
```

Eine *XGraphics*-Objekt erzeugen (Pendant zum GDI+ *Graphics*-Objekt):

```
        Dim gfx As XGraphics = XGraphics.FromPdfPage(page)
```

Schriftart erstellen:

```
        Dim font As New XFont("Verdana", 20, XFontStyle.BoldItalic)
```

Ausgabe:

```
        gfx.DrawString("Hallo, liebe Leser!", font, XBrushes.Black,
                       New XRect(0, 0, page.Width, page.Height), XStringFormats.Center)
```

Datei sichern und anzeigen:

```
        Const filename As String = "Test.pdf"
        document.Save(filename)
        Process.Start(filename)
    End Sub
End Class
```

Die Arbeit mit dem *XGraphics*-Objekt dürfte Sie sicher vor nicht allzu große Probleme stellen, weitere Fragen, die sicher auftreten werden, können Sie mit Hilfe der reichlichen Beispiele unter

LINK: `http://www.pdfsharp.net/wiki/PDFsharpSamples.ashx`

beantworten.

Für die Autoren hat sich die obige Lösung, auch wegen Ihrer umfassenden Komplexität, als (fast) optimale Lösung erwiesen.

8.8.9 Eine CSV-Datei erstellen

Die Extension *.csv* steht für *Comma-Separated Values* (oder auch *Character-Separated Values*) und bezeichnet eine Textdatei, die durch Trennzeichen – meistens Semikolon oder Komma – voneinander getrennte Werte enthält. Dies dient in der Regel dazu, eine Tabellenform nachbilden zu können, die im Original bestand und exportiert wurde.

Im vorliegenden Beispiel wollen wir zwei Varianten zeigen, wie man eine Tabelle aus einer Datenbank (*Nordwind.mdb*) in das CSV-Format exportieren kann.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie im Designer zwei Buttons:

```
<Window x:Class="WpfApplication1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="(CSV-Datei erzeugen)" Height="205" Width="525">
<StackPanel Margin="50">
<Button Content="Variante 1" Width="145" Height="30" Click="Button_Click_1"/>
<Button Content="Variante 2" Width="145" Height="30" Click="Button_Click_2"/>
</StackPanel>
</Window>
```

Allgemeiner Code

Kopieren Sie zunächst die Datenbank *Nordwind.mdb* in Ihren Projektordner.

Folgende zusätzliche Namespaces sind erforderlich:

```
Imports System.Data
Imports System.Data.OleDb
Imports System.IO
Imports System.Text
...
```

Die folgende Funktion gibt eine *DataTable* zurück, welche die Daten der Tabelle *Personal* enthält:

```
    Private Function loadData() As DataTable
       Dim conn As New OleDbConnection(
                           "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb")
       Dim cmdStr As String = "SELECT Vorname, Nachname, Funktion, Ort, Land FROM Personal"
       Dim da As New OleDbDataAdapter(cmdStr, conn)
       Dim dt As New DataTable()
       conn.Open()
       da.Fill(dt)
       Return dt
    End Function
```

Variante 1

Dies ist die klassische Version, wie sie evtl. für den Einsteiger transparenter ist:

```
Private Sub WriteDataTable(filePath As String, dt As DataTable, seperator As Char)
    Using sw As New StreamWriter(filePath, False, System.Text.Encoding.Default)
        Dim colCount As Integer = dt.Columns.Count

        For i As Integer = 0 To colCount - 1
            sw.Write(dt.Columns(i))
            If i < colCount - 1 Then
                sw.Write(seperator)
            End If
        Next

        sw.Write(sw.NewLine)

        For Each dr As DataRow In dt.Rows
            For i As Integer = 0 To colCount - 1
                sw.Write(dr(i).ToString())

                If i < colCount - 1 Then
                    sw.Write(seperator)
                End If
            Next
            sw.Write(sw.NewLine)
        Next
    End Using
End Sub
```

Variante 2

Deutlich kürzer und schneller ist diese Überladung, welche eine ganze Palette anspruchsvollerer Sprachfeatures nutzt (Generics, *StringBuilder*, Lambda-Ausdrücke) und in der bereits das Komma fest als Trennzeichen integriert ist:

```
Private Sub WriteDataTable(filePath As String, dt As DataTable)
    Dim sb As New StringBuilder()
    Dim columnNames As IEnumerable(Of String) =
            dt.Columns.Cast(Of DataColumn)().Select(Function(column) column.ColumnName)
    sb.AppendLine(String.Join(",", columnNames))
    For Each row As DataRow In dt.Rows
        Dim fields As IEnumerable(Of String) =
                                row.ItemArray.Select(Function(field) field.ToString())
        sb.AppendLine(String.Join(",", fields))
    Next
    File.WriteAllText(filePath, sb.ToString())
End Sub
```

Test

Füllen Sie die bereits vorhandenen Rahmencodes beider Eventhandler wie folgt aus:

Variante 1:

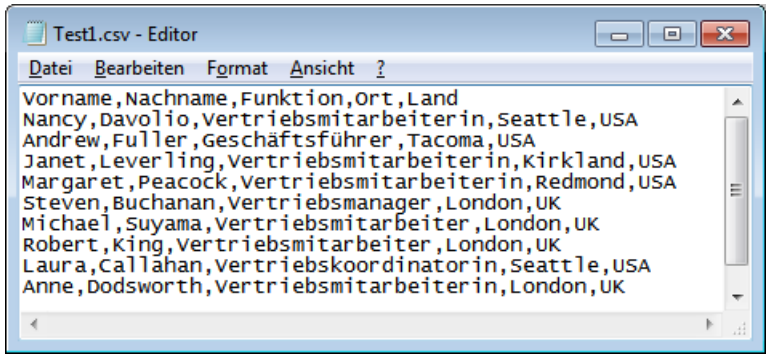
```
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        WriteDataTable("Test1.csv", loadData(), Convert.ToChar(","))
    End Sub
```

Variante 2:

```
   Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        WriteDataTable("Test2.csv", loadData())
    End Sub
...
End Class
```

Nach dem Programmstart klicken Sie einmal auf jeden der beiden Buttons.

Im */bin/Debug*-Unterverzeichnis Ihres Projektordners werden Sie jetzt die Dateien *Test1.csv* und *Test2.csv* vorfinden. Öffnen Sie diese mit einem Texteditor (*Notepad*), so dürfte sich Ihnen ein jeweils identischer Anblick bieten:



Bemerkungen

- Verwenden Sie Tabellen mit deutlich mehr Datensätzen, so werden Sie feststellen, dass Variante 1 viel langsamer als Variante 2 ist.
- Natürlich könnten Sie nach entsprechender Anpassung der Verbindungszeichenfolge auch auf andere Datenbanken bzw. Provider (SQL-Server) zugreifen.
- Wie Sie die erzeugte CSV-Datei weiter verarbeiten können, zeigt Ihnen das folgende Beispiel

  ► 8.8.10 Eine CSV-Datei mit LINQ lesen und auswerten

8.8.10 Eine CSV-Datei mit LINQ lesen und auswerten

Noch immer ist unter .NET die Praxis verbreitet, eine CSV-Datei zeilenweise einzulesen und dann die Daten unter Verwendung bestimmter Zeichenkettenfunktionen innerhalb von *For*-Schleifen zu extrahieren. LINQ bietet aber einen viel eleganteren Weg.

Wir wollen für unser vorliegendes Beispiel die in

► 8.8.9 Eine CSV-Datei erstellen

erzeugte CSV-Beispieldatei verwenden, welche Personaldaten aus *Nordwind.mdb* enthält.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und fügen Sie dem Formular lediglich eine *ListBox* hinzu:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CSV lesen" Height="350" Width="525">
    <Grid>
        <ListBox Name="ListBox1" />
    </Grid>
</Window>
```

LINQ-Abfrage 1: eine komplette Datei einlesen

Kopieren Sie die Datei *Test1.csv* in das *\bin\Debug*-Unterverzeichnis Ihres Projektordners.

Zunächst wollen wir keine Datensätze selektieren, sondern einfach nur alle Datensätze abfragen und mittels *For Each*-Statement ausgeben.

```
Imports System.IO

Class MainWindow
```

Der Test der LINQ-Abfrage erfolgt einfachheitshalber gleich im (von uns hinzuzufügenden) Konstruktor des Hauptfensters:

```
    Public Sub New()
        InitializeComponent()
```

Die LINQ-Abfrage:

```
        Dim query = From line In File.ReadAllLines("Test1.csv")
                    Let personRecord = line.Split(","c)
                    Select New With {
                    Key .Vorname = personRecord(0),
                    Key .Nachname = personRecord(1),
                    Key .Funktion = personRecord(2),
                    Key .Ort = personRecord(3),
                    Key .Land = personRecord(4)}
```

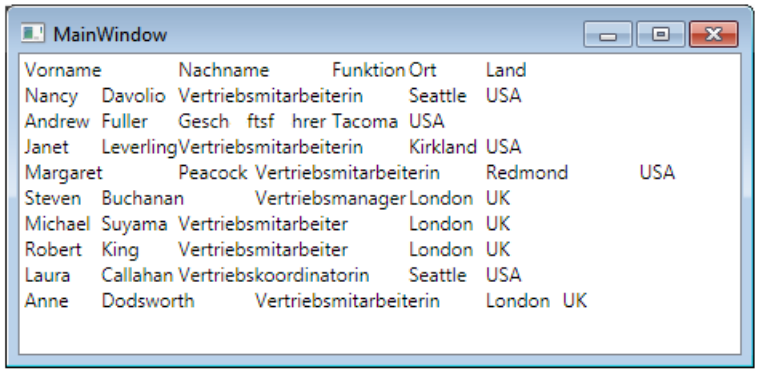
File.ReadAllLines() liefert ein Array mit den Zeilen der Datei, danach wird mittels *Split*-Funktion das Array am Komma getrennt.

Die Ausgabe:

```
        For Each item In query
          lb.Items.Add(item.Vorname & vbTab & item.Nachname & vbTab &
                       item.Ort & vbTab & item.Land)
        Next

    End Sub
End Class
```

Test

Unmittelbar nach Start der Anwendung sehen Sie bereits die Ergebnisse der Abfrage (siehe folgende Abbildung).



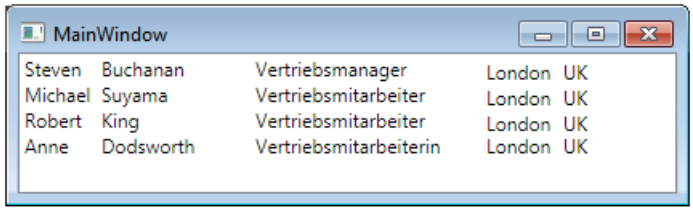
LINQ-Abfrage 2: Selektierte Datensätze lesen

Wenn zum Beispiel alle Personen aus UK ermittelt werden sollen, so ist obige LINQ-Abfrage durch die folgende zu ersetzen:

```
            Dim query = From pers In
           (From line In File.ReadAllLines("Test1.csv")
                  Let personRecord = line.Split(",")
                  Select New With {
                  Key .Vorname = personRecord(0),
                  Key .Nachname = personRecord(1),
                  Key .Funktion = personRecord(2),
                  Key .Ort = personRecord(3),
                  Key .Land = personRecord(4)})
               Where pers.Land = "UK"
               Select pers
```

Test

Das Ergebnis der Abfrage:



Bemerkungen

- Sie können nach Belieben mit weiteren LINQ-Abfragen experimentieren. Wenn Sie zum Beispiel die *Where*-Klausel ersetzen durch

  ```
  where pers.Funktion.Contains("manager")...
  ```

  dann selektieren Sie alle Personen, in deren *Funktion* die Bezeichnung "manager" enthalten ist.

- Beachten Sie, dass all diese Abfragen mit LINQ to Objects möglich sind, d.h., Sie brauchen dazu keine weiteren LINQ-Flavours wie LINQ to SQL oder LINQ to Entities zu bemühen.

8.8.11 Einen korrekten Dateinamen erzeugen

Einige Zeichen (" >, <, |, ...) sind in Dateinamen unzulässig. Die folgende Funktion überprüft den übergebenen Dateinamen und wandelt ihn gegebenenfalls in einen zulässigen Dateinamen um. Dabei werden unzulässige Zeichen durch ein "-" ersetzt:

```
Private Function ValidFilename(filename As String) As String
    For Each c As Char In System.IO.Path.GetInvalidFileNameChars()
        filename = filename.Replace(c, "-"c)
    Next
    Return filename
End Function
```

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie auf dem Stammelement eine *TextBox* (*Name=TextBox1*), einen *Button* und ein *Label* (*Name=Label1*).

Quellcode (MainWindow.xaml.vb)

```
Class MainWindow
```

Kopieren Sie hierhin den Code für die zu testende Funktion:

```
    Private Function ValidFilename(filename As String) As String
       ... ' siehe oben
    End Function
```

Test und Ergebnisanzeige:

```
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Label1.Content = ValidFilename(TextBox1.Text)
    End Sub
End Class
```

Test

Geben Sie in die *TextBox* einen beliebigen Dateinamen ein, nach dem Klick auf den Button erscheint ein "sauberer" Name.

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 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. Viele bereits bekannte Konzepte, wie z.B. der Zugriff auf den Vordergrund-Thread (Oberfläche), ändert sich auch bei Verwendung der *Task Parallel Library* nicht, 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*

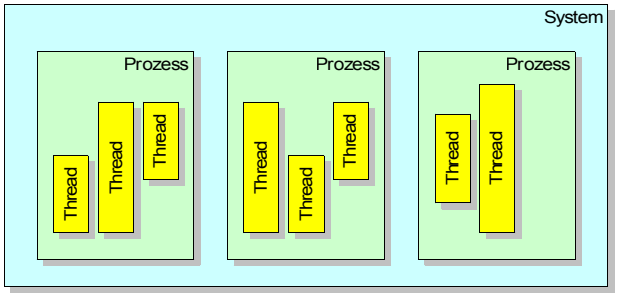
Im Folgenden wollen wir diese Begriffe im Zusammenhang erklären.

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 neuen *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 folgenden Kapitel 10 ein.

Die folgende Abbildung zeigt die grundsätzliche Aufteilung (unterschiedlich viele Threads können zu unterschiedlichen Zeiten innerhalb der Prozesse ausgeführt werden).



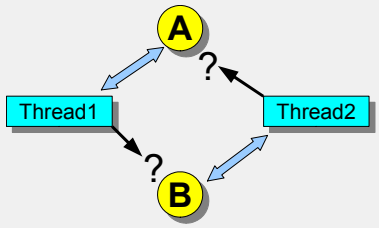
[1] Echte parallele Ausführung ist nur auf Mehrprozessorsystemen möglich.

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 schwer zu reproduzieren ist.

BEISPIEL 9.2: Racing

VB

Ein kleines Testprogramm (mit der neuen *Task Parallel Library*, siehe ab Seite 579) zeigt die Problematik eines "Wettrennens" am Beispiel der Ausgabekonsole.

```
Imports System.Threading
Imports System.Threading.Tasks

Module Module1
    Sub Main()
        Console.WriteLine("Start")
```

BEISPIEL 9.2: Racing

Zwei Tasks (Threads) erzeugen und starten:

```
        Dim thread1 = Task.Factory.StartNew(AddressOf MethodeA)
        Dim thread2 = Task.Factory.StartNew(AddressOf MethodeB)
```

Hier synchronisieren wir und warten bis **beide** Threads beendet sind:

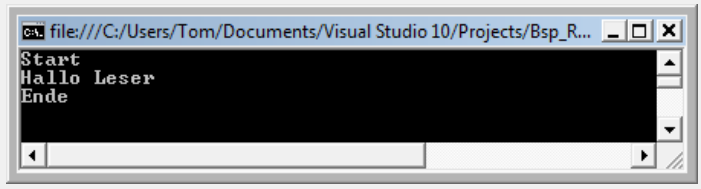
```
        Task.WaitAll(thread1, thread2)
        Console.WriteLine("Ende")
        Console.Read()
    End Sub
```

Die beiden "anspruchsvollen" Methoden für die jeweiligen Threads:

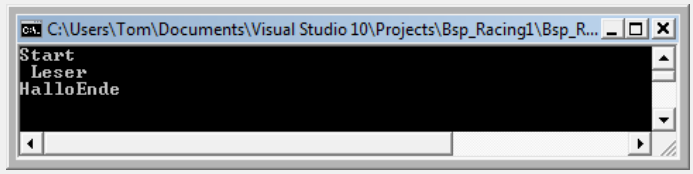
```
    Sub MethodeA()
        Console.Write("Hallo")
    End Sub

    Sub MethodeB()
        Console.WriteLine(" Leser")
    End Sub
End Module
```

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 Showstopper 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.

VB

```
...
Imports System.Threading
Imports System.Media

Public Class Form1
    Private myThread As Thread

    Private Sub Button1_Click(sender As Object, e As EventArgs) _
                                            Handles Button1.Click
```

Einen neuen Thread erzeugen (wir übergeben einen Delegaten, d h. die Adresse der Methode, die als eigener Thread ausgeführt werden soll):

```
        myThread = New Thread(AddressOf Berechne)
```

Wir starten den Thread:

```
        myThread.Name = "Test-Thread"
        myThread.IsBackground = True
        Debug.Print("Thread gestartet")
        myThread.Start(5)
```

Nach dem Aufruf der Methode *Start* wird die Programmausführung asynchron fortgesetzt, d h., noch vor dem Ende des Threads wird die nachfolgende Anweisung ausgeführt.

```
        Debug.Print("Nach Thread.Start")
    End Sub
```

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.

Die eigentliche Arbeit verrichtet die folgenden Methode:

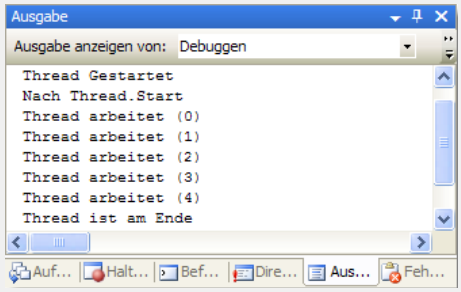
```
Private Sub Berechne(anzahl As Object)
```

Eine Schleife, um den Thread etwas warten zu lassen:

```
    For i As Integer = 0 To DirectCast(anzahl, Integer) - 1
        Thread.Sleep(1000)
        Debug.Print("Thread arbeitet ({0})", i)
        SystemSounds.Beep.Play()
    Next
    Debug.Print("Thread ist am Ende")
End Sub
```

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:

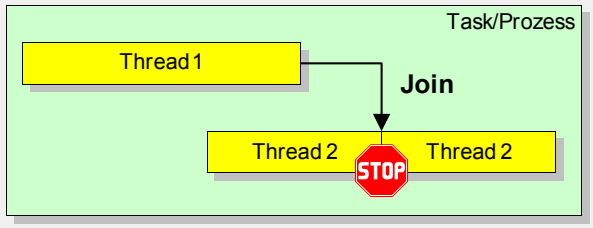


Ganz nebenbei werden Sie aus dem Lautsprecher mit Tönen gefoltert, aber so schlafen Sie beim Ausprobieren dieses Trivialbeispiels wenigstens nicht ein.

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.
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!

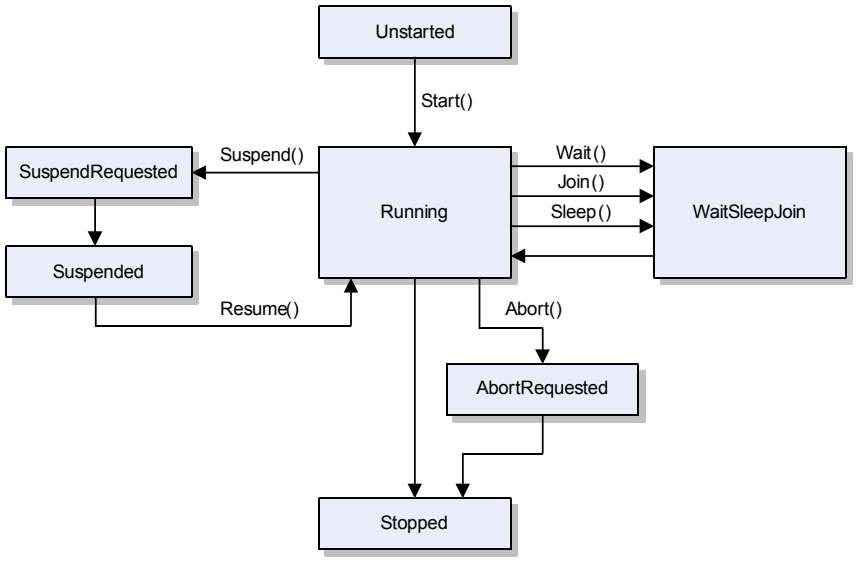
Methode	Beschreibung
Resume	Setzt die Ausführung des Threads fort, egal wie oft vorher *Suspend* aufgerufen wurde. Achtung: Diese Methode ist veraltet!
Interrupt	Unterbricht einen Thread, der sich im Wait-, Sleep- oder Join-Zustand befindet. Hierdurch wird eine *ThreadInterruptedException* ausgelöst, 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 VB-Compiler für Sie.

```
Private Sub button1_Click(ByVal sender As Object, ByVal e As EventArgs)
    Me.myThread = New Thread(New ParameterizedThreadStart(Me, CType(Me.Berechne, IntPtr)))
    Me.myThread.Name = "Test-Thread"
    Me.myThread.IsBackground = True
    Debug.Print("Thread Gestartet")
    Me.myThread.Start(5)
    Debug.Print("Nach Thread.Start")
End Sub
```

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.*
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.

Eine umfangreiche Anwendung, in welcher Sie den Einsatz der *Thread*-Klasse "spielend" lernen, finden Sie im Praxisbeispiel

► 9.10.1 Spieltrieb & Multithreading erleben

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 ThreadPool

VB

```
Imports System.Threading

Public Class Form1
    Public Sub Berechne(obj As Object)
        Dim n1 As Integer, n2 As Integer, n3 As Integer
        Debug.Print("-> Thread {0} gestartet", DirectCast(obj, Integer))
        For i As Integer = 0 To 9
            Thread.Sleep(100)
        Next
```

Abrufen der maximalen Anzahl von Threads:

```
        System.Threading.ThreadPool.GetMaxThreads(n3, n2)
```

Abrufen der noch verfügbaren Threads:

```
        System.Threading.ThreadPool.GetAvailableThreads(n1, n2)
        Debug.Print("  <- Thread {0} beendet! (frei {1} von {2})",
                                        DirectCast(obj, Integer), n1 + 1, n3)
    End Sub
```

Wir addieren 1, da der aktuelle Thread ja auch gleich beendet wird.

Das eigentliche Verwenden des Threadpools:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim n2 As Integer, n3 As Integer
```

BEISPIEL 9.4: Anwendung eines ThreadPool

Maximale Anzahl von Threads bestimmen:

```
        System.Threading.ThreadPool.GetMaxThreads(n3, n2)
```

Wir erzeugen ein paar mehr Threads als in den Threadpool passen:

```
        For i As Integer = 0 To (n3 + 10) - 1
            Debug.Print("Thread {0} in Warteschlange aufgenommen", i)
            System.Threading.ThreadPool.QueueUserWorkItem(AddressOf Berechne, i)
        Next
    End Sub
End Sub
```

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)
  <- 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 SyncLock

BEISPIEL 9.5: Wir erzeugen zwei Threads, die auf die gleiche Variable *summe* zugreifen. Ein Thread versucht zu inkrementieren, der andere zu dekrementieren.

VB

```
...
Imports System.Threading

Public Class Form1

    Public summe As Integer = 0
```

Die Thread-Methode:

```
    Public Sub Berechne(anzahl As Object)
        For i As Integer = 0 To 4
            Thread.Sleep(100)
```

Auch wenn die folgende Anweisung im VB-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 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 += DirectCast(anzahl, Integer)
            Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe)
        Next
    End Sub
```

Erzeugen zweier Threads:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim t1 As Thread = New Thread(AddressOf Berechne)
        t1.Name = "T1"
        Dim t2 As Thread = New Thread(AddressOf Berechne)
        t2.Name = "T2"
```

BEISPIEL 9.5: Wir erzeugen zwei Threads, die auf die gleiche Variable *summe* zugreifen. Ein Thread versucht zu inkrementieren, der andere zu dekrementieren.

```
        t1.Start(1)
        t2.Start(-1)
    End Sub
...
```

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 SyncLock

Unter Visual Basic bietet sich als einfachste Möglichkeit das *SyncLock*-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 *SyncLock*-Statement an) freigegeben wird.

BEISPIEL 9.6: Besser läuft es mit einem *SyncLock*-Statement:

VB

```
    Public Sub BerechneSicher(anzahl As Object)
        SyncLock Me
            For i As Integer = 0 To 4
                Thread.Sleep(100)
```

BEISPIEL 9.6: Besser läuft es mit einem *SyncLock*-Statement:

```
                summe += DirectCast(anzahl, Integer)
                Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe)
            Next
        End SyncLock
    End Sub
```

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 *SyncLock*-Statement eingeschlossen werden.

BEISPIEL 9.7: Bessere Lösung

VB

```
...
        For i As Integer = 0 To 4
            Thread.Sleep(100)
            SyncLock Me
                summe += DirectCast(anzahl, Integer)
                Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe)
            End SyncLock
        Next
...
```

BEISPIEL 9.7: Bessere Lösung

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 *SyncLock*-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 *SyncLock*-Statements nicht weit genug, verwenden Sie doch einen *Monitor*. Auch diese Klasse finden Sie 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)

VB

```
    Public Sub Beladen()
        Monitor.Enter(Me)
        Wartezeit = 0
        Ladung += 1
        myForm.FormRefresh("Schiff: Beladen (" & Ladung.ToString() & ")")
        If Ladung = 5 Then Transport()
        Monitor.Exit(Me)
    End Sub
```

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:

```
...
        Try
            Monitor.Enter(Me)
        Catch
        Finally
            Monitor.Exit(Me)
        End Try
...
```

HINWEIS: Auch wenn Sie ein *SyncLock*-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 *SyncLock*-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*

VB

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 bekannt. Doch nach dem fünften Schleifendurchlauf in *BerechneSicherMonitorA* soll diese Methode angehalten werden (Methode *Wait*), um zunächst der Methode *BerechneSicherMonitorB* die Vorfahrt zu lassen. 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 Sub BerechneSicherMonitorA(anzahl As Object)
    For i As Integer = 0 To 9
        Thread.Sleep(100)
        Monitor.Enter(Me)
        summe += DirectCast(anzahl, Integer)
```

Nach fünf Durchläufen warten wir zunächst (hier könnten Sie auch einen Zeitwert übergeben):

```
        If (i = 5) Then Monitor.Wait(Me)
        Debug.Print("Thread {0}: i = {1}  Summe = {2}", Thread.CurrentThread.Name,
                    i, summe)
```

BEISPIEL 9.9: Verwendung *Wait* und *Pulse*

```
            Monitor.Exit(Me)
        Next
    End Sub
```

Methode *BerechneSicherMonitorB:*

```
    Public Sub BerechneSicherMonitorB(anzahl As Object)
        For i As Integer = 0 To 99
            Thread.Sleep(100)
            Monitor.Enter(Me)
            summe += DirectCast(anzahl, Integer)
```

Nach 50 Durchläufen wird die Sperre in *BerechneSicherMonitorA* wieder aufgehoben:

```
            If (i = 50) Then Monitor.Pulse(Me)
            Debug.Print("Thread {0}: i = {1}  Summe = {2}", Thread.CurrentThread.Name,
                        i, summe)
            Monitor.Exit(Me)
        Next
    End Sub
```

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
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).
```

BEISPIEL 9.9: Verwendung *Wait* und *Pulse*

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* (*mutually exclusive*) für das Locking von Codeabschnitten an. Win32-Programmierern wird dieser Begriff sicher bekannt vorkommen, denn ein analoger Mechanismus war auch dort vorhanden.

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

VB

Mutex deklarieren:

```
Public Class Form1
```

BEISPIEL 9.10: Anwendung eines Mutex

```
    Private myMutex As Mutex = New System.Threading.Mutex()
...
    Public Sub BerechneMutex(anzahl As Object)
        myMutex.WaitOne()
        For i As Integer = 0 To 4
            Thread.Sleep(200)
            summe += DirectCast(anzahl, Integer)
            Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe)
        Next
        myMutex.ReleaseMutex()
    End Sub
```

Aufruf der Threads:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim t1 As Thread = New Thread(AddressOf BerechneMutex)
        Dim t2 As Thread = New Thread(AddressOf BerechneMutex)
        t1.Name = "T1" : t2.Name = "T2"
        t1.Start(1)
        t2.Start(-1)
    End Sub
```

Ergebnis

```
Thread T1: Summe = 1
Thread T1: Summe = 2
Thread T1: Summe = 3
Thread T1: Summe = 4
Thread T1: Summe = 5
Der Thread 0x754 hat mit Code 0 (0x0) geendet.
Thread T2: Summe = 4
Thread T2: Summe = 3
Thread T2: Summe = 2
Thread T2: Summe = 1
Thread T2: Summe = 0
Der Thread 0xc28 hat mit Code 0 (0x0) geendet.
```

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 allzu viel 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

VB

```
...
Imports System.Runtime.CompilerServices
...
    <MethodImpl(MethodImplOptions.Synchronized)>
    Public Sub Berechne(anzahl As Object)
        For i As Integer = 0 To 4
            Thread.Sleep(100)
            summe += DirectCast(anzahl, Integer)
            Debug.Print("Thread {0}: Summe = {1}", Thread.CurrentThread.Name, summe)
        Next
    End Sub
```

Obige Methode wird in keinem Fall durch andere Threadaufrufe unterbrochen, sondern immer komplett abgearbeitet.

Vergleichsbeispiel

Übertragen auf unsere "Bar"-Analogie 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.

VB

Die Semaphore erzeugen und initialisieren:

```
    Private sem As New Semaphore(3, 3)       ' 3 frei, 3 insgesamt verfügbar
...
```

Wir starten 10 Threads, von denen aber maximal 3 gleichzeitig zum Zuge kommen:

```
    Private Sub Button8_Click(sender As Object, e As EventArgs) Handles Button8.Click
```

BEISPIEL 9.12: Die Routine *BerechneMitSemaphore* soll von maximal drei Threads gleichzeitig genutzt werden.

```
        For i As Integer = 0 To 9
            Dim t As New Thread(AddressOf BerechneMitSemaphore)
            t.Name = "Thread" + i.ToString()
            t.Start()
        Next
    End Sub
```

Die eigentliche Routine:

```
    Public Sub BerechneMitSemaphore(anzahl As Object)
```

Nur wenn noch nicht 3 angemeldete Threads vorliegen, kommen wir hier weiter:

```
        sem.WaitOne()
        For i As Integer = 0 To 4
            Thread.Sleep(100)
            Debug.Print("Thread {0}: ", Thread.CurrentThread.Name)
        Next
```

Wir geben die Semaphore (bzw. einen Platz) wieder frei:

```
        sem.Release()
    End Sub
```

Ergebnis

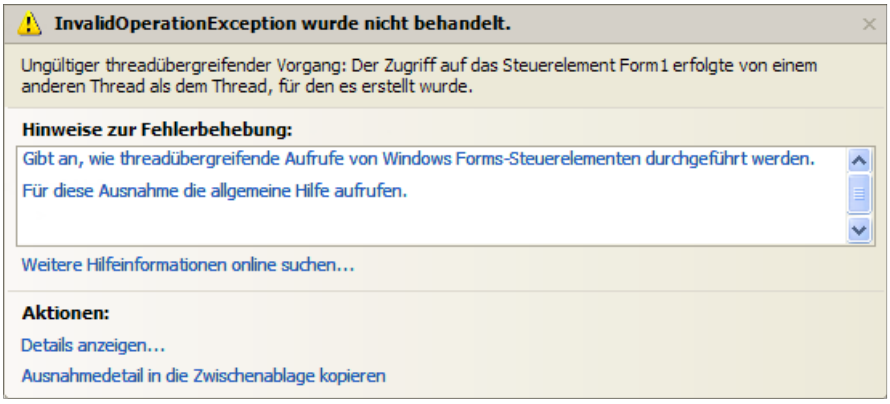
Die Ausgabe: Bei einem Blick ins Ausgabefenster werden Sie feststellen, dass sich maximal 3 Threads gleichzeitig abmühen.

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 neuen *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.

HINWEIS: 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* sollen Ergebnisse aus verschiedenen Threads angezeigt werden.

VB

Definition des Delegaten:

```
Public Delegate Sub MyAnzeigeDelegate(msg As String)
```

Hier wird der Wert in die *ListBox* eingetragen, dank aufrufender *Invoke*-Methode im Kontext des Vordergrund-Threads:

```
Public Sub myAnzeige(msg As String)
    ListBox1.Items.Add(msg)
End Sub
```

Unsere Callback-Funktion:

```
Public Sub BerechneMitAnzeige(anzahl As Object)
    SyncLock Me
        For i As Integer = 0 To 4
           Thread.Sleep(200)
           summe += 1
           Dim s As String = "Thread " & Thread.CurrentThread.Name &
                             ": Summe = " & summe.ToString
```

Zuweisen des Delegaten für die *Invoke*-Methode:

```
           ListBox1.Invoke(New MyAnzeigeDelegate(AddressOf myAnzeige), New Object() {s})
        Next
    End SyncLock
End Sub
```

Der Threadaufruf:

```
Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
    Dim t1 As Thread = New Thread(AddressOf BerechneMitAnzeige)
    t1.Name = "T" & Id.ToString
    Id += 1
    t1.Start()

End Sub
```

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

VB
```
    Public Delegate Sub MyAnzeigeDelegate(msg As String)

    Public Sub myAnzeige(msg As String)
        ListBox1.Items.Add(msg)
        Label1.Text = "Bla Bla"
        ProgressBar1.Value += 2
    End Sub

    Public Sub Anzeige(msg As String)
        Me.Invoke(New MyAnzeigeDelegate(AddressOf myAnzeige), New Object() {msg})
    End Sub
```

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*

VB
```
    Sub MeineAnzeige()
        If (ListBox1.InvokeRequired) Then
```

Falls nötig, rufen wir die Methode im Kontext des Controls erneut auf:

```
            ListBox1.Invoke(New MethodInvoker(AddressOf MeineAnzeige))
        Else
            ListBox1.Items.Add("Hello World!")
        End If
    End Sub
```

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

VB

```
...
Imports System.Threading
Imports System.Threading.Tasks

Class MainWindow
    Private id As Integer
```

Definition des Delegaten:

```
    Public Delegate Sub MyAnzeigeDelegate(msg As String)
```

Hier wird der Wert in die *ListBox* eingetragen, dank aufrufender *Invoke*-Methode im Kontext des Vordergrund-Threads:

```
    Public Sub myAnzeige(msg As String)
        ListBox1.Items.Add(msg)
    End Sub
```

Unsere Thread-Methode:

```
Public Sub BerechneMitAnzeige(anzahl As Object)
    SyncLock Me
       For i As Integer = 0 To 4
          Thread.Sleep(200)
          Dim s As String = ("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*:

```
          If ListBox1.Dispatcher.CheckAccess() Then
             ListBox1.Items.Add(s)
          Else
              ListBox1.Dispatcher.Invoke(New MyAnzeigeDelegate(AddressOf myAnzeige),
                                                                          New Object() {s})
          End If
       Next
   End SyncLock
End Sub
...
```

Thread erzeugen und starten:

```
    Private Sub Button1_Click(sender As Object, e As RoutedEventArgs) _
                                                Handles Button1.Click
        Dim t1 As New Thread(AddressOf BerechneMitAnzeige)
        t1.Name = "T" & id.ToString()
```

BEISPIEL 9.16: Verwendung von *Dispatcher.Invoke* in einer WPF-Anwendung

```
        id += 1
        t1.Start()
    End Sub
```

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

VB

```
...
Imports System.Threading

Public Class Form1
    Private myTimer As System.Threading.Timer
```

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 *Nothing*), die Wartezeit bis zum Start des Timers (*Timeout.Infinite* um den sofortigen Start zu verhindern) sowie das Zeitintervall in Millisekunden (oder als *TimeSpan*).

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        myTimer = New System.Threading.Timer(AddressOf MeinTimer, "Timer", 1000, 500)
    End Sub
```

BEISPIEL 9.17: Verwendung eines Timer-Threads

Die Callback-Routine (mit dem Parameter-Objekt):

```
Public Sub MeinTimer(obj As Object)
    Dim s As String = DirectCast(obj, String)
    ListBox1.Invoke(New MyAnzeigeDelegate(AddressOf myAnzeige), New Object() {s})
End Sub
```

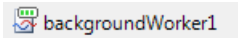
Anzeige in der *ListBox*:

```
Public Delegate Sub MyAnzeigeDelegate(msg As String)

Public Sub myAnzeige(msg As String)
    ListBox1.Items.Add(msg)
End Sub
```

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

VB

```
Imports System.Threading
Imports System.Diagnostics

Public Class Form1
```

BEISPIEL 9.18: Hintergrundberechnung mit Hilfe eines *BackgroundWorker*-Steuerelements

Die Hintergrund-Routine:

```
Private Sub BackgroundWorker1_DoWork(sender As Object,
                                     e As DoWorkEventArgs) Handles BackgroundWorker1.DoWork
    Dim res As Int32 = 0
```

Über *e.Argument* können Sie den Startparameter abfragen. Sie können beliebige Objekte verwenden, müssen diese jedoch typisieren:

```
    For i As Integer = 0 To DirectCast(e.Argument, Integer) - 1
```

Hat der Nutzer einen Abbruch angefordert (*CancelAsync*-Methode), können Sie dies über *CancellationPending* auswerten:

```
        If (TryCast(sender,System.ComponentModel.BackgroundWorker)).
                                        CancellationPending Then
            e.Cancel = True
            Exit For
        End If
        res += i
```

Hier produzieren wir etwas Luft:

```
        Thread.Sleep(100)
        Debug.Print("BackgroundWorker: " & res.ToString())
```

Soll ein Fortschritt angezeigt werden (*WorkerReportsProgress=True*), können Sie hier den Wert übergeben:

```
        TryCast(sender, System.ComponentModel.BackgroundWorker).ReportProgress(
                        i / e.Argument * 100)
    Next
```

Das Ergebnis lässt sich per *e.Result* gewinnen, auch hier sind beliebige Objekte möglich:

```
    e.Result = res
End Sub
```

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 lassen:

```
Private Sub BackgroundWorker1_ProgressChanged(sender As Object,
                                e As ProgressChangedEventArgs) _
                              Handles BackgroundWorker1.ProgressChanged
    ProgressBar1.Value = e.ProgressPercentage
End Sub
```

Wurde der Thread abgearbeitet oder findet ein Nutzerabbruch statt, wird das folgende Ereignis ausgeführt:

BEISPIEL 9.18: Hintergrundberechnung mit Hilfe eines *BackgroundWorker*-Steuerelements

```
Private Sub BackgroundWorker1_RunWorkerCompleted(sender As Object,
                            e As RunWorkerCompletedEventArgs) _
                                        Handles BackgroundWorker1.RunWorkerCompleted
```

Falls ein Fehler aufgetreten ist:

```
        If e.Error IsNot Nothing Then
            MessageBox.Show(e.Error.Message)
```

Oder es handelt sich um einen Nutzerabbruch:

```
        Else
            If e.Cancelled Then
                ListBox1.Items.Add("Nutzerabbruch")
```

Oder es ist alles in Ordnung und wir können das Ergebnis auswerten:

```
            Else
                ListBox1.Items.Add("ERGEBNIS:" & e.Result.ToString())
            End If
        End If
        Button1.Enabled = True
        Button2.Enabled = False
        progressBar1.Value = 0
    End Sub
```

Last, but not least müssen wir den Thread auch noch starten:

```
    Private Sub Button1_Click(sender As Object, e As System.EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Button2.Enabled = True
        BackgroundWorker1.RunWorkerAsync(100)
    End Sub
```

Hier können Sie ein Objekt als Parameter an den Thread übergeben.

Ist Ihnen nach Abbruch zumute, hilft die Methode *CancelAsync*:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        BackgroundWorker1.CancelAsync()
        Button2.Enabled = False
    End Sub
```

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

VB
```
Dim res As IAsyncResult = stateobj.fs.BeginRead(stateobj.Bytes, 0,
                  Convert.ToInt32(stateobj.fs.Length), AddressOf 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

VB
```
        While Not res.IsCompleted
            ...
        End While
        Dim BytesGelesen As Integer = fs.EndRead(res)
```

Die Rückrufmethode

Die Rückrufmethode wird nach dem Ende der asynchronen Methode aufgerufen. Sie muss folgendem Muster entsprechen:

```
Public Sub <Methodenname>(asyncResult As IAsyncResult)
```

Über *asyncResult.AsyncState* können Sie auf das eventuell übergebene Nutzer-Objekt zugreifen:

```
    Dim state As myState = DirectCast(asyncResult.AsyncState, myState)
```

In dieser Methode rufen Sie im Allgemeinen das Funktionsergebnis per *End<Methodenname>* ab. Dafür benötigen Sie den Parameter *asyncResult*:

```
    state.BytesGelesen = state.fs.EndRead(asyncResult)
```

Wie Sie das Entwurfsmuster konkret einsetzen, sollen Ihnen die folgenden vier kurzen 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

VB

Den notwendigen Namespace einbinden:

```
Imports System.IO

Public Class Form1
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Add("Starte Lese-Vorgang ...")
```

Einen *FileStream* erzeugen:

```
        Dim fs As FileStream = New FileStream("01.jpg", FileMode.Open,
                                    FileAccess.Read, FileShare.Read, 1000, True)
```

Das Byte-Array anlegen:

```
        Dim Bytes As Byte() = New Byte(fs.Length) {}
```

Den asynchronen Aufruf starten:

```
        Dim res As IAsyncResult = fs.BeginRead(Bytes, 0, Convert.ToInt32(fs.Length),
                                               Nothing, Nothing)
```

BEISPIEL 9.21: Polling

Der Rückgabewert ist vom Typ *IAsyncResult*. Mit einer Schleife fragen wir den Status solange ab, bis die Operation abgeschlossen ist:

```
        While Not res.IsCompleted
            ListBox1.Items.Add("IsCompleted = " & res.IsCompleted.ToString())
        End While
        ListBox1.Items.Add("IsCompleted = " & res.IsCompleted.ToString())
```

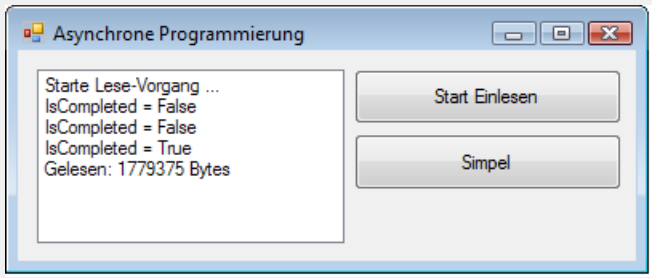
Mit einem synchronen Dateizugriff hätten wir mit weniger Aufwand dasselbe erreicht, aber es geht hier ja um asynchrone Aufrufe und die Abfrage könnte ja auch per Timer etc. erfolgen.

Bestimmen der Anzahl gelesener Bytes:

```
        Dim BytesGelesen As Integer = fs.EndRead(res)
        ListBox1.Items.Add("Gelesen: " & BytesGelesen.ToString() & " Bytes")
    End Sub
End Class
```

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

VB

Wir schreiben das Beispiel aus dem vorhergehenden Abschnitt so um, dass wir per Callback auf das Ende reagieren können.

```
Public Sub Fertig(asyncResult As IAsyncResult)
    ' hier ist die Operation zu Ende
End Sub
 ...
fs.BeginRead(Bytes, 0, fs.Length, AddressOf Fertig, Nothing)
...
```

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

VB

```
Imports System.IO

Public Class Form1
```

Die Hilfsklasse:

```
    Public Class myState
        Public Bytes As Byte()
        Public fs As FileStream
        Public BytesGelesen As Integer
    End Class
```

Die neue Callback-Routine:

```
    Public Sub Fertig(asyncResult As IAsyncResult)
```

BEISPIEL 9.23: Callback mit Parameterübergabe

Hilfsobjekt typisieren und verwenden:

```
        Dim state As myState = DirectCast(asyncResult.AsyncState, myState)
        state.BytesGelesen = state.fs.EndRead(asyncResult)
        Me.Invoke(New FormRefreshDelegate(AddressOf FormRefresh), state)
        state.fs.Close()
    End Sub
```

Das Starten des asynchronen Dateizugriffs:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Add("Starte Lese-Vorgang ...")
```

Hier erzeugen wir unser Hilfsobjekt:

```
        Dim stateobj As myState = New myState()
```

Die nötigen 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:

```
        Dim res As IAsyncResult = stateobj.fs.BeginRead(stateobj.Bytes, 0,
                    Convert.ToInt32(stateobj.fs.Length), AddressOf Fertig, stateobj)
    End Sub
```

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

VB

```
Public Class Form1
```

Unsere Routine, die auf die Steuerelemente zugreifen kann:

```
    Public Sub FormRefresh(obj As myState)
        ListBox1.Items.Add("CallBack: --- Fertig ---")
        ListBox1.Items.Add("CallBack: Gelesen: " & obj.BytesGelesen.ToString() & " Bytes")
    End Sub
```

BEISPIEL 9.24: Zugriff auf die Programmoberfläche

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 Sub FormRefreshDelegate(obj As myState)
```

Und zum Schluss ändern wir noch die Callback-Routine:

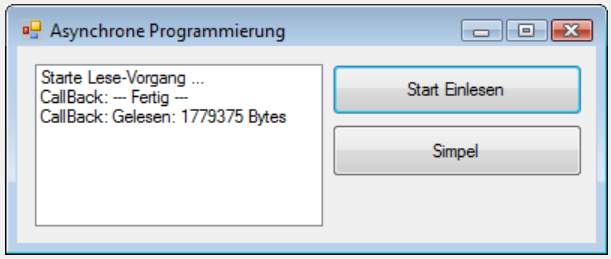
```
Public Sub Fertig(asyncResult As IAsyncResult)
    Dim state As myState = DirectCast(asyncResult.AsyncState, myState)
    state.BytesGelesen = state.fs.EndRead(asyncResult)
```

Übergabe des Delegates und des Hilfsobjekts:

```
    Me.Invoke(New FormRefreshDelegate(AddressOf FormRefresh), state)
    state.fs.Close()
End Sub
```

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 eine synchrone Methode *Berechne* zur Verfügung stellen soll:

BEISPIEL 9.25: Beispielklasse definieren

VB

```
Imports System.Threading
Imports System.Media
```

Die Klassendefinition:

```
Class LangeBerechnung
```

Eine Eigenschaft für das Endergebnis der "Berechnung":

```
    Public Ergebnis As Integer
```

Die eigentliche Funktion:

```
    Public Function Berechne(anzahl As Integer) As Integer
        Ergebnis = 0
```

Wir produzieren etwas Laufzeit und eine akustische Rückmeldung:

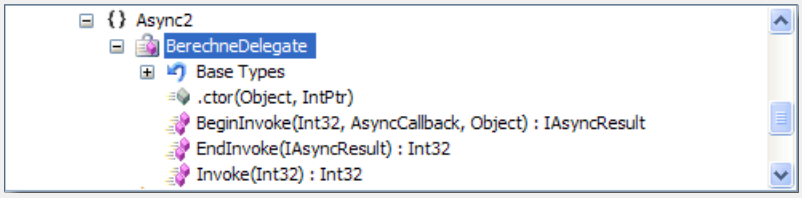
```
       For i As Integer = 0 To anzahl - 1
            Thread.Sleep(1000)
            SystemSounds.Beep.Play()
            Ergebnis += i
        Next
        Return Ergebnis
    End Function
End Class
```

Die Grundlage für den asynchronen Aufruf einer Methode ist die Definition und Instanziierung eines entsprechenden Delegaten!

Deshalb an dieser Stelle:

```
Public Delegate Function BerechneDelegate(anzahl As Integer) As Integer
```

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:



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

VB

Mit dem Button-Klick wird folgender Code abgearbeitet:

```
ListBox1.Items.Clear()
ListBox1.Items.Add("Start")
```

Instanz bilden:

```
Dim lb As LangeBerechnung = New LangeBerechnung()
```

Delegate instanziieren und Methode zuweisen:

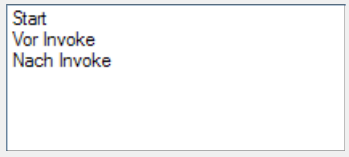
```
Dim dg As BerechneDelegate = New BerechneDelegate(AddressOf lb.Berechne)
ListBox1.Items.Add("Vor Invoke")
```

Hier erfolgt der asynchrone Aufruf, die *BeginInvoke*-Methode hat der Compiler für uns erzeugt:

```
dg.BeginInvoke(5, Nothing, Nothing)

ListBox1.Items.Add("Nach Invoke")
```

Ergebnis

Ein Test lässt uns lediglich anhand der *ListBox*-Einträge (diese werden ohne Unterbrechung sofort ausgegeben) ...



... 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

VB

Wir deklarieren zunächst eine Callback-Funktion (Parameter nicht vergessen):

```
Private Sub MyCallback(res As IAsyncResult)
    Me.Invoke(New anzeigedelegate(AddressOf anzeige), New Object() {"Callback: Fertig"})
End Sub
```

BEISPIEL 9.27: Modifizieren des Vorgängerbeispiels durch Einfügen von Callback- und Anzeige-Funktion

Wie Sie sehen, müssen wir wieder etwas Aufwand treiben, um Bildschirmausgaben zu produzieren (per *Invoke* wechseln wir in den Vordergrundthread):

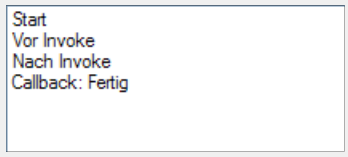
```
Public Delegate Sub anzeigedelegate(msg As String)
Public Sub anzeige(msg As String)
    ListBox1.Items.Add(msg)
End Sub
```

Unsere Vorbereitungen für den Aufruf der *BeginInvoke*-Methode sind fast gleich geblieben:

```
...
        Dim lb As New LangeBerechnung()
        Dim dg As BerechneDelegate = AddressOf lb.Berechne
        ListBox1.Items.Add("Vor Invoke")
        dg.BeginInvoke(5, AddressOf MyCallback, Nothing)
...
```

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

VB

Anpassen der Callback-Routine:

```
Private Sub MyCallback(res As IAsyncResult)
    If res.AsyncState IsNot Nothing Then
       Me.Invoke(New anzeigedelegate(AddressOf anzeige),
                                    New Object() {"Callback: Fertig"})
```

BEISPIEL 9.28: Rückgabewert per Eigenschaft

Rückgabewert per Eigenschaft:

```
          Me.Invoke(New anzeigedelegate(AddressOf anzeige), New Object() {"Ergebnis" &
                (TryCast(res.AsyncState, LangeBerechnung)).Ergebnis.ToString()})
       Else
          Me.Invoke(New anzeigedelegate(AddressOf anzeige),
                                    New Object() {"Callback: Fertig"})
       End If
    End Sub
```

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:

```
Dim lb As LangeBerechnung = New LangeBerechnung()
Dim dg As BerechneDelegate = New BerechneDelegate(AddressOf lb.Berechne)
ListBox1.Items.Add("Vor Invoke")
dg.BeginInvoke(5, AddressOf 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*

VB

```
Dim lb As LangeBerechnung = New LangeBerechnung()
Dim dg As BerechneDelegate = New BerechneDelegate(AddressOf lb.Berechne)
ListBox1.Items.Add("Vor Invoke")
dg.BeginInvoke(5, AddressOf MyCallback2, dg)
```

Die Callback-Methode:

```
Private Sub MyCallback2(res As IAsyncResult)
  If res.AsyncState IsNot Nothing Then
     Me.Invoke(New anzeigedelegate(AddressOf anzeige),
              New Object() {"Callback: Fertig"})
     Dim Result As Integer = (DirectCast(res.AsyncState,
                             BerechneDelegate)).EndInvoke(res)
     Me.Invoke(New anzeigedelegate(AddressOf anzeige), New Object() {"Ergebnis" &
                                                                    Result.ToString})
  End If
End Sub
```

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 haben Sie bereits gemerkt, 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 damit mancher auch die Lust am asynchronen Programmieren.

Dass ist auch den Microsoft-Entwicklern nicht verborgen geblieben, und so wurden mit Visual Basic 2015 zwei neue Schlüsselwörter eingeführt:

- 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 Windows Runtime (siehe dazu ab Kapitel 18) treibt dass Ganze auf die Spitze: Um die Oberfläche von Rucklern und Wartezeiten zu befreien, sind alle WinRT-Aufrufe mit einer Laufzeit von mehr als 50ms asynchron programmiert.

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.

HINWEIS: Haben Sie einen langsamen Computer empfehlen wir Ihnen die obigen Schleifen etwas zu kürzen. Besitzer einer SSD sollten die Werte erhöhen.

BEISPIEL 9.30: Synchrones Speichern von Daten

VB

```
...
Imports System.IO
```

Unsere Dummy-Daten zum Speichern (500 KB-String):

```
      Private dummy As String = "#".PadLeft(5000000, "#")
...
      Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
         ListBox1.Items.Clear()
         ListBox1.Items.Add("Start")
```

Datei erzeugen:

```
         Using writer As StreamWriter = File.CreateText("VieleDaten.txt")
           For i As Integer = 0 To 1000
```

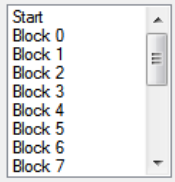
Wir speichern den Dummy-String mehrfach mit der *Write*-Methode:

```
               writer.Write(System.DateTime.Now.ToString & dummy)
               ListBox1.Items.Add("Block " + i.ToString)
           Next
         End Using
         ListBox1.Items.Add("FERTIG")
```

Datei wieder löschen, sonst ist Ihre Festplatte bald voll:

```
          File.Delete("VieleDaten.txt")
       End Sub
...
```

Ergebnis



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

VB

Wir markieren unsere Methode als asynchron:

```
   Private Async Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
       ListBox1.Items.Clear()
       ListBox1.Items.Add("Start")
```

BEISPIEL 9.31: Asynchrones Speichern

```
        Using writer As StreamWriter = File.CreateText("VieleDaten.txt")
            For i As Integer = 0 To 1000
```

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.Add("Block " & i.ToString)
            Next
        End Using
        ListBox1.Items.Add("FERTIG")
        File.Delete("VieleDaten.txt")
    End Sub
```

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 wäre 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 obiges Beispiel nochmals genauer an.

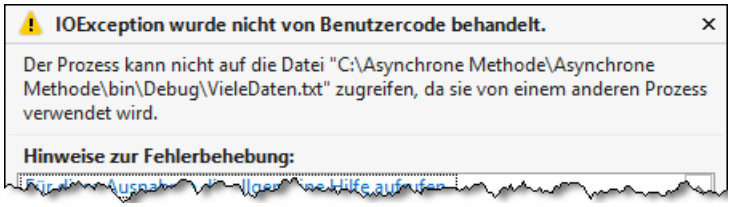
Die gesamte Methode *Button1_Click* wird im Vordergrund-Thread ausgeführt (vorteilhaft für die uneingeschränkte Interaktion mit der Userinterface), der *WriteAsync*-Aufruf läuft hingegen in einem extra Thread und gibt damit Rechenzeit für den Vordergrund (und damit für das Userinterface) frei. Der *Await*-Operator ermöglicht uns eine synchrone Verarbeitung innerhalb der asynchronen Methode.

Doch was passiert eigentlich, wenn wir auf den *Await*-Operator verzichten?

In diesem Fall würde die gesamte Methode *Button1_Click* 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 potenzielle 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 (*Enabled=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

VB

Der ausgelagerte, asynchrone Code in einer extra Routine:

```
Async Private Sub DateiSpeichern()
   Using writer As StreamWriter = File.CreateText("VieleDaten.txt")
    For i As Integer = 0 To 1000
        Await writer.WriteAsync(System.DateTime.Now.ToString & dummy)
        ListBox1.Items.Add( "Block " & i.ToString)
    Next
  End Using
End Sub
```

Der Aufruf obiger Methode:

```
Private Sub Button2_Click(sender As Object, e As RoutedEventArgs)
    ListBox1.Items.Clear()
    ListBox1.Items.Add("Start")
    DateiSpeichern()
    ListBox1.Items.Add("FERTIG")         ' logischer Fehler
    File.Delete("VieleDaten.txt")        ' Fehler beim Datei-Zugriff
End Sub
```

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 *Datei-Speichern* noch läuft. Wie stand es doch 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:

```
Private Async Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    ListBox1.Items.Clear()
    ListBox1.Items.Insert(0, "Start")
    Await DateiSpeichern()
    ListBo 'DateiSpeichern' gibt keine Task zurück und kann nicht erwartet werden. Ziehen Sie eine Änderung in eine Async-Funktion in Betracht.
    File.D
End Sub
```

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(Of T)* zurückgeben.

BEISPIEL 9.33: Diese Funktion kann mit *Await* genutzt werden

VB

```
...
    Private Async Function DateiSpeichern2() As Task
        Using writer As StreamWriter = File.CreateText("VieleDaten.txt")
            For i As Integer = 0 To 1000
                Await writer.WriteAsync(System.DateTime.Now.ToString & dummy)
                ListBox1.Items.Add("Block " & i.ToString)
            Next
        End Using
    End Function
```

Erst jetzt lässt sich die Methode korrekt verwenden:

```
Private Async Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    ...
    Await DateiSpeichern2()
    ...
End Sub
```

Um den Rückgabewert, d h. den Thread, kümmert sich in diesem Fall der Compiler (eine *Return*-suchen Sie vergeblich).

9.9.3 Eigene asynchrone Methoden entwickeln

Sicher nicht ganz uninteressant ist die Frage, wie Sie Ihren eigenen Code so umschreiben/konvertieren können, dass Sie (oder auch Anwender Ihrer Libraries) die neuen *Async*-Features nutzen können.

BEISPIEL 9.34: Alter, synchroner Code

VB

```
...
Imports System.Threading
Imports System.Threading.Tasks
```

Ausgangspunkt soll folgende Routine sein:

```
    Private Function MeineLangeBerechnung() As Integer
        Dim res As Integer = 0
        For i As Integer = 0 To 50
            res += i
```

Zeitbedarf im aktuellen Thread simulieren:

```
            Thread.Sleep(200)
            Debug.WriteLine("Rechne + " & i.ToString
        Next i
```

BEISPIEL 9.34: Alter, synchroner Code

```
        Return res
    End Function
```

Sie sehen schon, die obige Routine müht sich, möglichst langsam zu sein und gibt einen Integer-Wert zurück.

Der Aufruf wird erwartungsgemäß die Oberfläche für eine gewisse Zeit in den Tiefschlaf versetzen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    ListBox1.Items.Clear()
    ListBox1.Items.Add("Start")
    Dim summe As Integer = MeineLangeBerechnung()
    ListBox1.Items.Add("Summe = " & summe.ToString)
    ListBox1.Items.Add("Ende")
End Sub
```

Wir wollen es nun besser machen:

BEISPIEL 9.35: Neuer, asynchroner Code

VB

```
...
    Private Async Function MeineLangeBerechnungAsync1() As Task(Of Integer)
        Dim res As Integer = 0
        For i As Integer = 0 To 50
            res += i
```

Der simulierte Zeitbedarf landet jetzt automatisch in einem extra Thread, die Programmoberfläche hat damit genügend Zeit, Nutzereingaben zu verarbeiten:

```
            Await Task.Delay(200)        ' Zeitbedarf simulieren
            ListBox1.Items.Add("Rechne + " & i.ToString)
        Next i
        Return res
    End Function
```

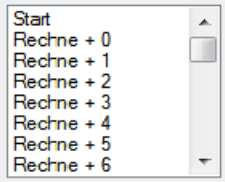
Eine kleine Änderung im obigem Code haben Sie vielleicht schon bemerkt: In der Schleife der Berechnungsfunktion brauchen Sie jetzt nicht mit *Debug.WriteLine* zu arbeiten, Sie können auch Ausgaben direkt in die aktuelle Oberfläche schreiben, und das ohne einen Threadwechsel!

Die asynchrone Verwendung:

```
Private Async Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    ListBox1.Items.Clear()
    ListBox1.Items.Add("Start")
    Dim summe As Integer = Await MeineLangeBerechnungAsync1()
    ListBox1.Items.Add("Summe = " & summe.ToString)
    ListBox1.Items.Add("Ende")
End Sub
```

BEISPIEL 9.35: Neuer, asynchroner Code

Ergebnis



Das war doch gar nicht so schlimm – mit geringsten Änderungen haben Sie asynchron ausführbaren Code produziert, ohne sich mit Threads herumärgern zu müssen!

Doch was ist, wenn Sie nicht in Ihren alten Libraries herumfrickeln wollen und trotzdem die *Async*-Features unterstützen wollen? Auch das ist möglich, in diesem Fall kapseln Sie einfach den ganzen alten Code (bzw. den Methodenaufruf dorthin) in einen extra Thread:

BEISPIEL 9.36: Alten synchronen Code kapseln

VB

```
...
    Public Function MeineLangeBerechnungAsync2() As Task(Of Integer)
```

Die alte Methode *MeineLangeBerechnung()* wird in einen neuen Thread mittels Lambda-Ausdruck eingebunden:

```
        Return Task.Factory.StartNew(Of Integer)(Function() MeineLangeBerechnung())
    End Function
...
```

Der asynchrone Aufruf:

```
    Private Async Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    ...
    Dim summe As Integer = Await MeineLangeBerechnungAsync2()
    ...
    End Sub
```

Ausgerüstet mit diesem Grundwissen dürfte nun einer Erweiterung Ihrer Libraries um asynchrone Methoden nichts mehr im Wege stehen.

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 einem 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 muss sein und hilft auch in einem solchen Fall, die Langeweile zu vertreiben!

HINWEIS: Für alle 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:

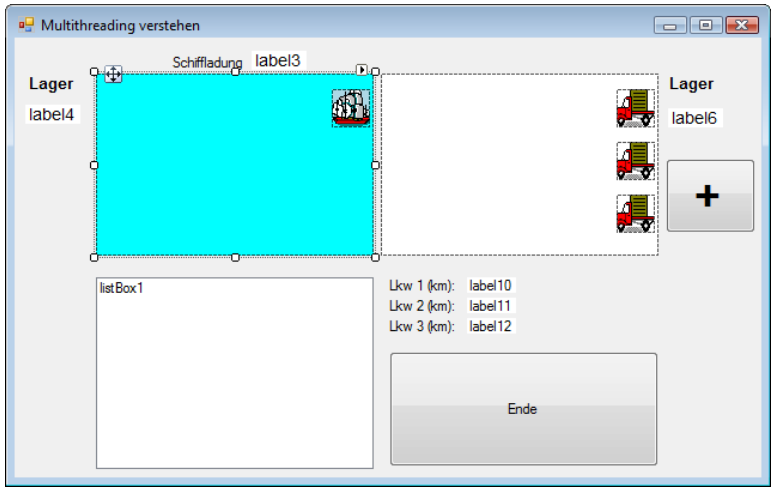
- LKW
- Schiff
- *Controller* (Lager-Controller)

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.

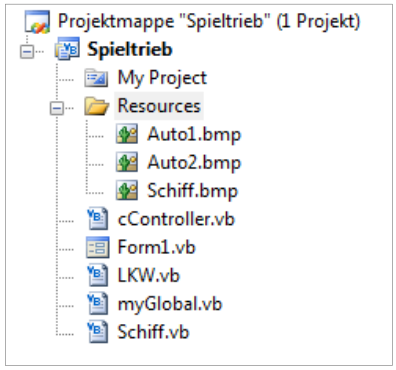
Oberfläche

Auf das Startformular *Form1* einer Windows Forms-Anwendung setzen Sie zwei *Panels*, einige *PictureBox*en sowie diverse *Labels* und *Buttons* entsprechend folgender Abbildung.



HINWEIS: Achten Sie darauf, dass "LKWs" und "Schiff" direkt in die *Panel*s 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 zugeordneten 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()*.

```
Imports System.Text
Imports System.Threading
Imports System.Drawing
Imports System.Drawing.Imaging

Class cLKW
```

Die Fahrzeugnummer:

```
    Public Nr As Integer
```

Die aktuelle Position:

```
    Public Pos As Integer
```

Der Kilometerstand:

```
    Public kmStand As Integer
```

Der Thread:

```
    Public myThread As Thread
```

Die Bilder:

```
    Private Img1 As Image
    Private Img2 As Image
```

Das aktuelle Bild:

```
    Public Image As Image
```

Ein Verweis auf das aufrufende Formular:

```
    Private myForm As Form1
```

Grundzustand initialisieren und *Thread*-Objekt erzeugen:

```
    Public Sub New(nr As Integer, myf As Form1)
        nr = nr
        kmStand = 0
        Pos = 280
        myForm = myf
        myThread = New Thread(AddressOf Fahren)
        myThread.IsBackground = True
        Img1 = My.Resources.Auto1
        Img2 = My.Resources.Auto2
        Image = Img1
    End Sub
```

Die spätere Thread-Methode:

```
    Public Sub Fahren()
```

Bild und Position zuweisen:

```
Image = Img1
Pos = 280
```

Die Fahrt beginnt:

```
For posi As Integer = 280 To 1 Step -2
```

Damit es nicht ganz so schnell geht, werden ein paar Pausen eingelegt:

```
    Thread.Sleep(30)
    Pos = posi
    kmStand += 1
```

Hier rufen wir die zentrale Anzeigeroutine in *Form1* auf:

```
    myForm.FormRefresh("")
Next
```

HINWEIS: Obige Methode greift nicht direkt auf die Steuerelemente von *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 myGlobal.Schiff.myThread.IsAlive Then
    myGlobal.Schiff.myThread.Join()
End If
```

Ist der Thread des Schiffs aktiv[1], können wir davon ausgehen, dass das Schiff unterwegs ist, andernfalls liegt es im Hafen und wir können es beladen:

```
myGlobal.Schiff.Beladen()
```

Rückfahrt (Bildwechsel und langsam nach rechts verschieben):

```
Image = Img2
For posi As Integer = 0 To 279 Step 2
    Thread.Sleep(20)
    Pos = posi
    kmStand += 1
    myForm.FormRefresh("")
Next
```

Letzte Statusmeldung für den LKW ausgeben, dann ist der Thread am Ende:

```
    myForm.FormRefresh("Lkw_" & Nr.ToString() & ": Fahrtende")
End Sub
```

[1] Um den Thread abfragen zu können, muss das Objekt auch vorhanden sein. Deshalb werden die Threads schon in den Konstruktoren der Objekte erzeugt.

Ruft der Controller die Methode *Abtransport* auf, wird eine Kiste umgeladen (Lagerbestand verringert sich), die Änderung wird angezeigt und der Thread gestartet:

```
    Public Sub Abtransport()
        myGlobal.Controller.Lagerbestand -= 1
        myForm.FormRefresh("Lkw_" & Nr.ToString() & ": Fahrtbeginn")
        myThread = New Thread(AddressOf Fahren)
        myThread.IsBackground = True
        myThread.Start()
    End Sub
End Class
```

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. Hierfür bietet sich ein *CallbackTimer* an.

```
Imports System.Text
Imports System.Threading

Class Schiff
```

Die Eigenschaften:

```
    Public Ladung As Integer
    Public Transportiert As Integer
    Private Wartezeit As Integer
    Public myForm As Form1
    Public pos As Integer
```

Timer und *Thread*:

```
    Public tim As Timer
    Public myThread As Thread
```

Der Konstruktor:

```
    Public Sub New(mf As Form1)
```

Position festlegen, *Thread* und *Timer* erzeugen:

```
        pos = 200
        myThread = New Thread(AddressOf Me.Main)
        myThread.IsBackground = True
        tim = New Timer(AddressOf OnTimer, Nothing, 0, 3000)
        Wartezeit = 0
```

Verweis auf *Form1* abspeichern (für die Anzeige der Daten):

```
        myForm = mf
        Transportiert = 0
    End Sub
```

Die Hauptroutine (Thread-Methode):

```
    Private Sub Main()
```

Benutzerschnittstelle aktualisieren:

```
        myForm.FormRefresh("Schiff: Abfahrt")
```

Nach links fahren:

```
        For posi As Integer = 200 To 1 Step -2
            Thread.Sleep(100)
            pos = posi
            myForm.FormRefresh("")
        Next
```

Im Hafen die Ladung löschen:

```
        myForm.FormRefresh("Schiff: Ladung löschen")
        While Ladung > 0
            Thread.Sleep(400)
```

Absichern:

```
            Monitor.Enter(Me)
            Ladung -= 1
            Transportiert += 1
            myForm.FormRefresh("")
            Monitor.Exit(Me)
        End While
```

Rückfahrt:

```
        For posi As Integer = 0 To 199 Step 2           'Rückfahrt
            Thread.Sleep(60)
            pos = posi
            myForm.FormRefresh("")
        Next
        Wartezeit = 0
        myForm.FormRefresh("Schiff: Warten ...")
    End Sub
```

Starten des Threads:

```
    Public Sub Transport()
        myThread = New Thread(AddressOf Main)
        myThread.IsBackground = True
        myThread.Start()
    End Sub
```

Das Schiff wird durch die LKWs "beladen":

```
    Public Sub Beladen()
        Monitor.Enter(Me)
        Wartezeit = 0
```

```
        Ladung += 1
        myForm.FormRefresh("Schiff: Beladen (" & Ladung.ToString() & ")")
```

Fünf Kisten an Bord, dann Abtransport:

```
        If Ladung = 5 Then Transport()
        Monitor.Exit(Me)
    End Sub
```

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 Sub OnTimer(state As Object)
```

Absichern des Abschnitts, sonst kommen uns noch andere Threads in die Quere:

```
        Monitor.Enter(Me)
```

Ist der eigene Thread inaktiv (Hafen!) verlängert sich die Wartezeit:

```
        If Not Me.myThread.IsAlive Then Wartezeit += 1
```

Ist die Wartezeit abgelaufen und mehr als eine Kiste vorhanden:

```
        If (Wartezeit > 9) And (Ladung > 0) Then
```

Abfahrt:

```
            myForm.FormRefresh("Schiff: Zeit abgelaufen")
            Wartezeit = 0
            Transport()
        End If
```

Der sichere Abschnitt wird verlassen:

```
        Monitor.Exit(Me)
    End Sub
End Class
```

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.

```
Imports System.Text
Imports System.Threading

Class cController
    Public Lagerbestand As Integer
    Public myThread As Thread
```

Wir sind am Leben:

```
Public IsAlive As Boolean
Private myForm As Form1
```

Das Objekt startet bereits im Konstruktor seinen Endlos-Thread:

```
Public Sub New(mf As Form1)
    myForm = mf
    myThread = New Thread(AddressOf Main)
```

Wir sind nicht ganz so wichtig, deshalb etwas weniger Rechenzeit:

```
    myThread.Priority = ThreadPriority.BelowNormal
    myThread.IsBackground = True
    myThread.Start()
    IsAlive = True
End Sub
```

Unser "Hauptprogramm":

```
Private Sub Main()
```

Ein paar Aktualisierungen des Formulars:

```
    myForm.FormRefresh(String.Empty)
    myForm.FormRefresh("Controller: Gestartet")
    Dim i As Integer = 0
```

Die Endlosschleife:

```
    While True And IsAlive
```

Es ist was im Lager:

```
        While Lagerbestand > 0
```

Einen der drei LKWs aussuchen[1]:

```
            If i > 2 Then i = 0
```

Ist der LKW nicht unterwegs (kein laufender Thread), dann beladen:

```
            If Not myGlobal.LKWs(i).myThread.IsAlive Then
                myForm.FormRefresh("Controller: Lkw beladen")
                Thread.Sleep(200)
                ' Ladezeit
                myGlobal.LKWs(i).Abtransport()
            Else
```

Sonst warten wir etwas (0,5 Sekunden) auf diesen LKW:

```
                myGlobal.LKWs(i).myThread.Join(500)
```

[1] Bei den späteren Tests werden Sie feststellen, dass die LKWs durch diese Auswahl recht gleichmäßig belastet werden, kein LKW ist im Dauereinsatz.

Ist er jetzt da, dann beladen:

```
                If Not myGlobal.LKWs(i).myThread.IsAlive Then
                    myForm.FormRefresh("Controller: Lkw beladen")
                    Thread.Sleep(200)
                    ' Ladezeit
                    myGlobal.LKWs(i).Abtransport()
                End If
            End If
            i += 1
        End While
```

Hier "ruht" sich der Controller etwas aus (Mittagsschlaf):

```
            Thread.Sleep(100)
        End While
        myForm.FormRefresh("Controller: Gestoppt")
    End Sub
End Class
```

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 MyGlobal

Wie Sie sicher bemerkt haben, rufen die Objekte/Klassen untereinander Methoden auf und greifen auf zentrale Objekte zu. Dies lässt sich zum Beispiel mit einigen statischen Membern realisieren:

```
...
Class myGlobal
    Public Shared Controller As cController
    Public Shared Schiff As Schiff
    Public Shared LKWs As cLKW() = New cLKW(3) {}
End Class
```

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.

```
Imports System.Threading

Public Class Form1
```

Für den *Invoke*-Aufruf von *MyRefresh* (auf die Steuerelemente können wir nur aus dem Vordergrund-Thread zugreifen) brauchen wir einen Delegate:

```
    Private Delegate Sub MyRefreshDelegate(msg As String)
```

Die "sichere" Methode für die Anzeige:

```
Private Sub MyRefresh(msg As String)
```

Nochmal absichern:

```
    SyncLock Me
```

Handelt es sich um eine Textmeldung, dann nur die *ListBox* aktualisieren:

```
        If msg <> "" Then
            ListBox1.Items.Add(msg)
        Else
```

Sonst die allgemeinen Daten abfragen und setzen:

```
            Label3.Text = myGlobal.Schiff.Ladung.ToString()
            Label4.Text = myGlobal.Schiff.Transportiert.ToString()
            Label6.Text = myGlobal.Controller.Lagerbestand.ToString()
            Label10.Text = myGlobal.LKWs(0).kmStand.ToString()
            Label11.Text = myGlobal.LKWs(1).kmStand.ToString()
            Label12.Text = myGlobal.LKWs(2).kmStand.ToString()
            PictureBox2.Image = myGlobal.LKWs(0).Image
            PictureBox3.Image = myGlobal.LKWs(1).Image
            PictureBox4.Image = myGlobal.LKWs(2).Image
            PictureBox1.Left = myGlobal.Schiff.pos
            PictureBox2.Left = myGlobal.LKWs(0).Pos
            PictureBox3.Left = myGlobal.LKWs(1).Pos
            PictureBox4.Left = myGlobal.LKWs(2).Pos
        End If
    End SyncLock
End Sub
```

Für den Aufruf aus den Threads:

```
Public Sub FormRefresh(msg As String)
```

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:

```
            Me.Invoke(New MyRefreshDelegate(AddressOf MyRefresh), New Object() {msg})
    Catch
    End Try
End Sub
```

Und hier erzeugen wir die Objekte (nicht im Konstruktor, sonst fehlt für den Controller das Fensterhandle):

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    PictureBox1.Image = My.Resources.Schiff
```

```
        myGlobal.LKWs(0) = New cLKW(0, Me) :  myGlobal.LKWs(1) = New cLKW(1, Me)
        myGlobal.LKWs(2) = New cLKW(2, Me) :  myGlobal.Schiff = New Schiff(Me)
        myGlobal.Controller = New cController(Me)
    End Sub
```

Das ist unser ganzer Eingriff ins System, wir füllen das Lager mit Kisten:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        myGlobal.Controller.Lagerbestand += 1
    End Sub
```

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 Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Die LKW-Threads werden "brutal" zerstört:

```
        If myGlobal.LKWs(0).myThread IsNot Nothing Then myGlobal.LKWs(0).myThread.Abort()
        If myGlobal.LKWs(1).myThread IsNot Nothing Then myGlobal.LKWs(1).myThread.Abort()
        If myGlobal.LKWs(2).myThread IsNot Nothing Then myGlobal.LKWs(2).myThread.Abort()
```

Beim Schiff bevorzugen wir die sanfte Variante:

```
        myGlobal.Schiff.tim.Dispose()
        If myGlobal.Schiff.Ladung > 0 Then
            myGlobal.Schiff.Transport()
            MessageBox.Show("Bitte erneut versuchen, Schiff muss noch arbeiten ...")
            Return
        End If
        If myGlobal.Schiff.myThread IsNot Nothing Then myGlobal.Schiff.myThread.Abort()
```

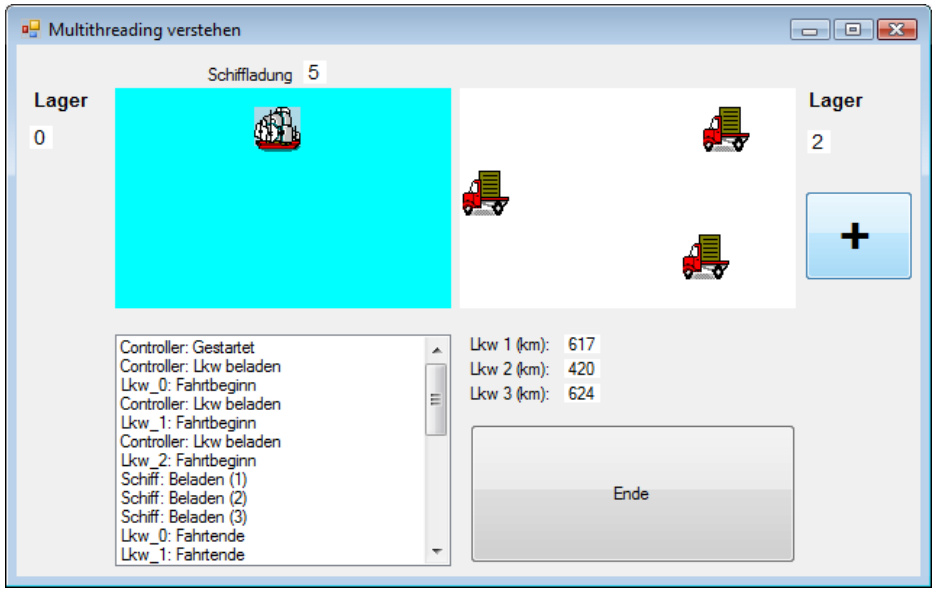
Unseren Controller beenden wir ganz sauber:

```
        myGlobal.Controller.IsAlive = False
        While myGlobal.Controller.IsAlive
            Thread.Sleep(200)
        End While
        Close()
    End Sub
End Class
```

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

- Verbesserungen und Erweiterungen sind natürlich immer möglich. Insbesondere der direkte Zugriff auf die jeweiligen Threads ist nicht sonderlich empfehlenswert. Auch die Verwendung eines Thread-Pools (z.B. für die LKWs) wäre denkbar.
- 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.
- Einen "moderneren" Lösungsansatz mit den TPL-Sprachkonstrukten bietet das Beispiel 10.9 am Ende des nachfolgenden Kapitels.

9.10.2 Prozess- und Thread-Informationen gewinnen

Dieses Universalbeispiel demonstriert Ihnen nicht nur die Verwendung der *Process*- und *ProcessThread*-Objekte in VB, 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*.

Quellcode

```
Public Class Form1
```

Unsere Haupt-Referenz auf die Process-Klasse (*System.Diagnostics*-Namespace):

```
Private ProcObj As Process = New Process()    ...
```

Einige Ergänzungen im *Load*-Ereignis des Formulars:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles Me.Load
    With ListView1.Columns
        .Add("Base Name", 110, HorizontalAlignment.Left)
        .Add("Modules", 70, HorizontalAlignment.Right)
        .Add("Full Path", 150, HorizontalAlignment.Left)
        .Add("PID", 40, HorizontalAlignment.Right)
        .Add("CPU Time", 70, HorizontalAlignment.Right)
        .Add("Mem Usage", 80, HorizontalAlignment.Right)
        .Add("Thread ID", 85, HorizontalAlignment.Right)
        .Add("State", 100, HorizontalAlignment.Right)
        .Add("WaitReason", 258, HorizontalAlignment.Left)
        .Add("CPU Time", 70, HorizontalAlignment.Right)
    End With
```

Der Aufruf unserer "Haupt-Methode" *enumProcs()*, die wir im Anschluss implementieren werden:

```
    Me.enumProcs()
End Sub
```

Die *enumProc()*-Methode ermittelt alle laufenden Prozesse:

```
Public Sub enumProcs()
    Dim lvItem As ListViewItem
    Dim ts As System.TimeSpan
```

Laufende Prozesse besorgen und in einem Array ablegen:

```
    Dim allProcs As Process() = Process.GetProcesses()
```

Alle Prozesse und ihre Beschreibung in *ListView1* ausgeben:

```
    For i As Integer = 0 To allProcs.Length - 1
        lvItem = Me.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())
            Try
                ts = allProcs(i).TotalProcessorTime
                lvItem.SubItems.Add(String.Format("{0:00}", ts.TotalHours) & ":" &
                                    String.Format("{0:00}", ts.Minutes) & ":" &
                                    String.Format("{0:00}", ts.Seconds))
```

```
                Catch
                    lvItem.SubItems.Add("Err")
                End Try
                lvItem.SubItems.Add((allProcs(i).WorkingSet64 / 1000).ToString() & "K")
                Label1.Text = "Prozesse: " & allProcs.Length.ToString()
            End Try
        Next
    End Sub
```

Diese Methode ermittelt alle Threads zu einem laufenden Prozess:

```
    Public Sub EnumThreads(ProcID As Integer)
        Dim ts As System.TimeSpan
        Dim lvItem As ListViewItem
```

Den *ListView*-Inhalt löschen:

```
        ListView2.Items.Clear()
```

Den Prozess referenzieren (mittels seiner ID):

```
        Dim aProc As Process = Process.GetProcessById(ProcID)
        Try
```

Anzeige der Anzahl von Threads:

```
            Label2.Text = "Threads: " & aProc.Threads.Count.ToString()
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
```

Alle Threads des Prozesses durchlaufen:

```
        For Each aThread As ProcessThread In aProc.Threads
            lvItem = ListView2.Items.Add(aThread.Id.ToString())
            Select Case aThread.ThreadState
                Case ThreadState.Initialized
                    lvItem.SubItems.Add("Initialized")
                Case ThreadState.Ready
                    lvItem.SubItems.Add("Ready")
                Case ThreadState.Running
                    lvItem.SubItems.Add("Running")
                Case ThreadState.Standby
                    lvItem.SubItems.Add("Standby")
                Case ThreadState.Terminated
                    lvItem.SubItems.Add("Terminated")
                Case ThreadState.Transition
                    lvItem.SubItems.Add("In Transition")
                Case ThreadState.Unknown
                    lvItem.SubItems.Add("Unkwown")
                Case ThreadState.Wait
                    lvItem.SubItems.Add("Waiting")
            End Select
```

Falls der Thread im Wartezustand ist, soll eine Info ausgegeben werden:

```
            If aThread.ThreadState = ThreadState.Wait Then
                lvItem.SubItems.Add(ListReason(aThread.WaitReason))
            Else
                lvItem.SubItems.Add("N/A")
            End If
```

TimeSpan-Objekt abholen:

```
            ts = aThread.TotalProcessorTime
```

Zeit formatieren:

```
            lvItem.SubItems.Add(String.Format("{0:00}", ts.TotalHours) & ":" &
                                String.Format("{0:00}", ts.Minutes) & ":" &
                                String.Format("{0:00}", ts.Seconds))
        Next
    End Sub
```

Die folgende Methode dechiffriert lediglich die *ThreadWaitReason*-Enumeration:

```
    Private Function ListReason(waitingReason As System.Diagnostics.ThreadWaitReason) _
                                                                          As String
        Dim s As String = ""
        Select Case waitingReason
            Case ThreadWaitReason.EventPairHigh
                s = "Waiting For Event Pair High"
            Case ThreadWaitReason.EventPairLow
                s = "Waiting For Event Pair Low"
            Case ThreadWaitReason.ExecutionDelay
                s = "Execution Delay"
            Case ThreadWaitReason.Executive
                s = "Waiting for Scheduler"
            Case ThreadWaitReason.FreePage
                s = "Waiting For Free Virtual Mem. Page"
            Case ThreadWaitReason.LpcReceive
                s = "Waiting For A Local Proc. Call To Arrive"
            Case ThreadWaitReason.LpcReply
                s = "Waiting For A Reply To A Local Proc. Call"
            Case ThreadWaitReason.PageIn
                s = "Waiting For Virtual Mem. Page To Arrive In Memory"
            Case ThreadWaitReason.PageOut
                s = "Waiting For Virtual Mem. Page To Write To Disk"
            Case ThreadWaitReason.Suspended
                s = "Execution Suspended"
            Case ThreadWaitReason.SystemAllocation
                s = "Waiting For A System Allocation"
            Case ThreadWaitReason.Unknown
                s = "Waiting For Unknown Reason"
            Case ThreadWaitReason.UserRequest
                s = "Waiting For A User Request"
```

```
            Case ThreadWaitReason.VirtualMemory
                s = "Waiting For Virtual Memory"
        End Select
        Return s
    End Function
```

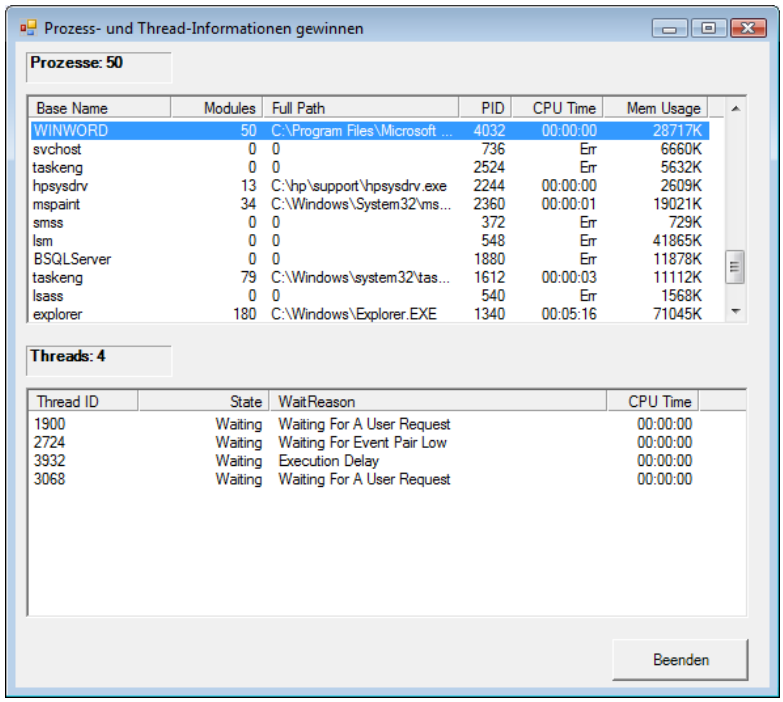
Ein neuer Prozess wird angeklickt:

```
    Private Sub ListView1_SelectedIndexChanged(sender As Object, e As EventArgs) _
                                               Handles ListView1.SelectedIndexChanged
        Try
            EnumThreads(Convert.ToInt32(ListView1.SelectedItems(0).SubItems(3).Text))
        Catch
        End Try
    End Sub
    ...
End Class
```

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.

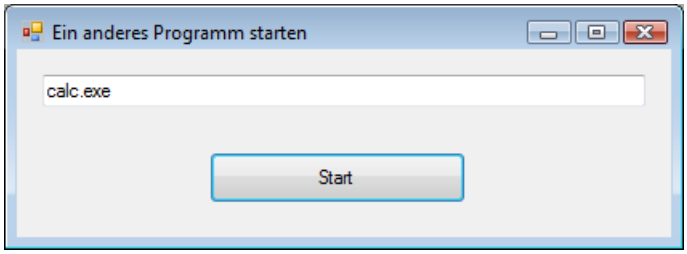


9.10.3 Ein externes Programm starten

Möchten Sie aus Ihrer .NET-Anwendung heraus andere Prozesse[1], d h. Programme, starten, bieten sich Ihnen unter Visual Basic 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

```
Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Erzeugen eines *Process*-Objekts:

```
        Dim proc As Process = New Process()
```

Parametrieren (welche Anwendung soll gestartet werden) und Aufruf:

```
        proc.StartInfo.FileName = TextBox1.Text
        proc.Start()
```

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()
        MessageBox.Show("Fertig")
    End Sub
```

HINWEIS: Möchten Sie die Anwendung bzw. den Prozess asynchron starten, lassen Sie einfach die Methode *WaitForExit()* weg.

[1] Hier ist wirklich von Prozessen und nicht von Threads die Rede.

Test

Geben Sie in die *TextBox* einen Anwendungsnamen (z.B. *Calc.exe*, *MSPaint.exe*, ...) ein und klicken Sie auf die "Start"-Schaltfläche:

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

```
VB System.Diagnostics.Process.Start("notepad.exe","c:\test.txt")
```

BEISPIEL 9.38: Eine Webseite im Explorer öffnen

```
VB System.Diagnostics.Process.Start("http://www.microsoft.com")
```

BEISPIEL 9.39: Eine E-Mail erzeugen

```
VB System.Diagnostics.Process.Start("mailto:max_musterman@nirgendwo.de")
```

Bemerkung

In diesem Beispiel haben wir Ihnen lediglich die grundsätzliche Methodik zum Starten von externen Anwendungen vorgestellt, Visual Basic 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.

Kapitel 10

Die Task Parallel Library

Nachdem Sie sich im vorhergehenden Kapitel 9 bereits intensiv mit den Möglichkeiten der asynchronen Programmierung und deren Grundlagen vertraut gemacht haben, wollen wir uns nun ausgiebig mit den praktischen Möglichkeiten der neuen **Task Parallel Library** beschäftigen.

10.1 Überblick

Sicher stellt sich Ihnen jetzt die Frage, warum Sie sich überhaupt mit den neuen 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 neuen Sprachkonstrukte *Parallel.For* und *Parallel.ForEach* besprechen wir ab Seite 585 im Detail.

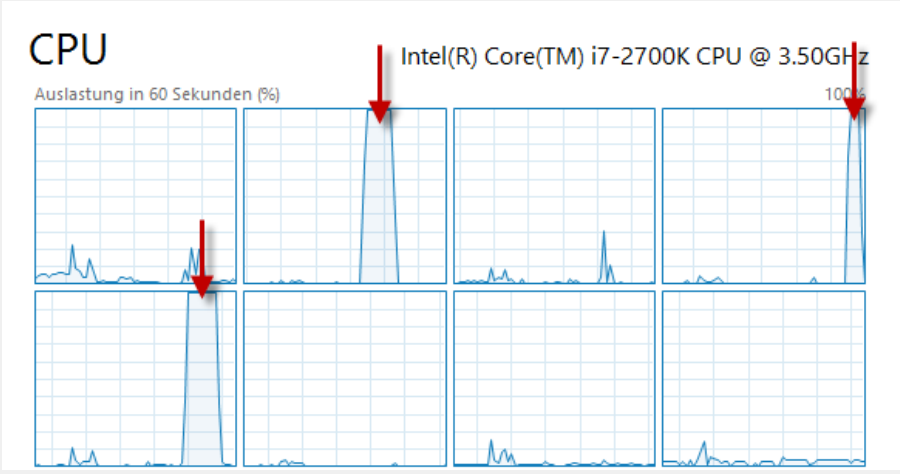
BEISPIEL 10.1: Unterschiede bei der Verarbeitung einer Schleife

VB

Die konventionelle Version mit einer ganz normalen *For*-Schleife:

```
Dim res As Double = 0
For i As Integer = 0 To 3999999
    res += (Math.Sin(New Random().NextDouble()) * Math.Acos(
                New Random().NextDouble()))
Next i
```

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".

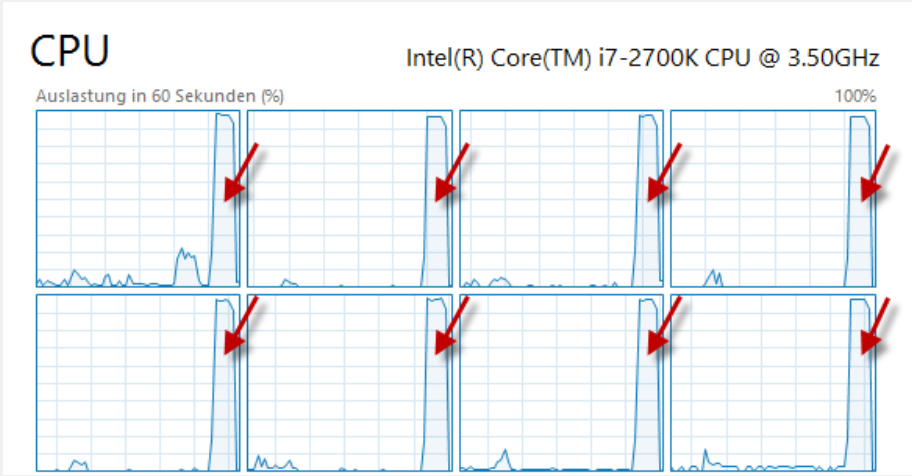
Die parallelisierte Version (zu den Details später mehr):

```
Dim res As Double = 0
Parallel.For(0, 4000000, Sub(i) res += (Math.Sin(New Random().NextDouble()) *
                         Math.Acos(New Random().NextDouble())))
```

Die Laufzeit beträgt jetzt nur noch 6.128 ms (rund ein Viertel) bei nahezu 100% Prozessorlast

BEISPIEL 10.1: Unterschiede bei der Verarbeitung einer Schleife

und gleichmäßiger Verteilung auf alle Kerne:



Je mehr Kerne bzw. Prozessoren, desto schneller kann jetzt die Aufgabe realisiert werden.

Ein Quadcore-Prozessor dürfte die obige Aufgabenstellung noch etwas mehr beschleunigen, können in diesem Fall doch die Threads auf 4 Kerne verteilt 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 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 neu berechnet werden. Hier führt schon der Ersatz des bisherigen *For* durch ein *Parallel.-*

[1] Mehr dazu unter *http //de.wikipedia.org/wiki/Amdahlsches_Gesetz*

For zu einer wesentlichen Beschleunigung der **Berechnung**. Beachten Sie jedoch auch in diesem Fall, 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 nötige 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 neuen 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.

---

[1] Sie ahnen es sicher, es handelt sich um die *Task Parallel Library*.

[2] *Common Language Runtime* = NET-Laufzeitumgebung

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 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 (Datenbankinitalisierung, Abfrage von Webdiensten, Laden von Grafiken, Durchsuchen des Dateisystems etc.).

SYNTAX: `Parallel.Invoke(ParamArray actions As Action())`

Übergeben Sie einfach eine Reihe von Methodennamen bzw. nutzen Sie die Möglichkeit, gleich Lambda-Ausdrücke aufzurufen. Doch jetzt zur praktischen Verwendung.

10.2.1 Aufrufvarianten

BEISPIEL 10.2: Parallele Verarbeitung von drei Methoden

VB

Namespace einbinden:

```
Imports System.Threading
Imports System.Threading.Tasks
```

Die drei Methoden definieren (ohne sinnvolle Aufgabe):

```
Private Sub AufgabeA()
    Thread.Sleep(5000)
End Sub

Private Sub AufgabeB()
    Thread.Sleep(5000)
End Sub

Private Sub AufgabeC()
    Thread.Sleep(5000)
End Sub
```

Der eigentliche Aufruf:

```
Private Sub Form1_Load(sender As Object, e As EventArgs)  Handles MyBase.Load
    MessageBox.Show("Methoden parallel aufrufen ...")
    Parallel.Invoke(AddressOf AufgabeA, AddressOf AufgabeB, AddressOf AufgabeC)
    MessageBox.Show("Alle Methoden abgearbeitet!")
End Sub
```

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

VB

```
...
    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

VB
```
...
        Parallel.Invoke(
              Sub() Thread.Sleep(3000),
              Sub() Thread.Sleep(3000),
              Sub() Thread.Sleep(3000))
...
```

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 neuen Sprachkonstrukt angelangt.

10.3 Verwendung von Parallel.For

Und hier haben wir es auch schon mit einer der interessantesten Neuerungen in .NET-Framework zu tun. Das *Parallel.For*-Konstrukt bietet neben *Parallel.ForEach* eine konkurrenzlos einfache Möglichkeit, 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, body As Action(Of Integer))`

Ein kleines Beispiel soll Ihnen das Grundprinzip demonstrieren.

BEISPIEL 10.5: Parallele Verarbeitung eines Arrays

VB

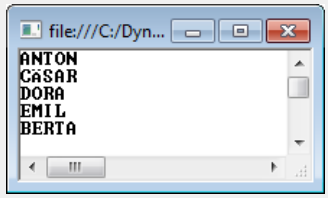
```
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
        Dim Vornamen() As String = {"anton", "berta", "cäsar", "dora", "emil", "friedrich"}
        Parallel.For(0, 5, Sub(i) Console.WriteLine(Vornamen(i).ToUpper()))
        Console.ReadKey()
    End Sub
```

Wie Sie sehen, steht dem Lambda-Ausdruck ein Parameter *i* zur Verfügung, über den der betreffende Index identifiziert werden kann.

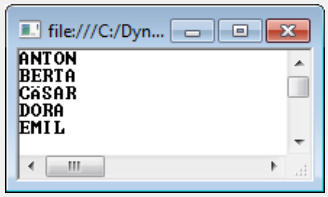
Der Code des obigen Lambda-Abschnitts wird gleichmäßig auf alle Kerne aufgeteilt, je mehr es davon gibt, umso besser.

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:

VB

```
...
        For i As Integer = 0 To 5
            Console.WriteLine(Vornamen(i).ToUpper())
        Next
...
```

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, body As Action(Of Integer, ParallelLoopState) )`

BEISPIEL 10.7: "Schleifenabbruch" realisieren

VB

```
Imports System.Threading
Imports System.Threading.Tasks
...
   Sub Main()
        Dim Vornamen() As String = {"anton", "berta",
                                    "cäsar", "dora", "emil", "friedrich"}
        Parallel.For(0, 5, Sub(i, loopstate)
```

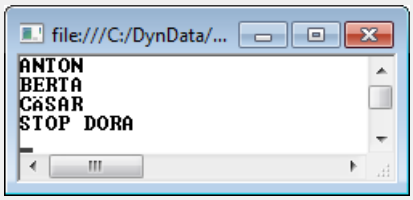
Vorzeitiges Ende, wenn in einer Zelle der Name "dora" gefunden wird:

```
                               If Vornamen(i) = "dora" Then loopstate.Stop()
                               If loopstate.IsStopped Then Console.Write("STOP ")
                               Console.WriteLine(Vornamen(i).ToUpper())
                           End Sub)
    End Sub
...
```

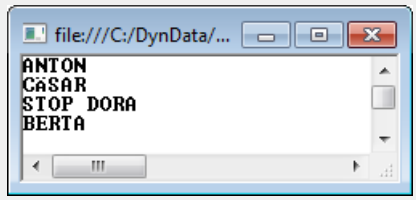
Es ist nicht sicher, dass nach der Verarbeitung des aktuellen Threads nicht doch noch ein anderer Thread ausgeführt wird, wie es auch die folgenden Abbildungen zeigen.

Ergebnis

Die optimale Version:



Ein Problemfall:



[1] Das muss nicht unbedingt auf unser "anspruchsvolles" Beispiel zutreffen, etwas mehr Logik sollte schon in den betreffenden Abschnitten parallelisiert werden.

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*

VB

```
Dim res = Parallel.For(0, 5, Sub(i, loopstate)
                                 If Vornamen(i) = "dora" Then loopstate.Stop()
                                 If loopstate.IsStopped Then Console.Write("STOP ")
                                 Console.WriteLine(Vornamen(i).ToUpper())
                             End Sub)
```

Wurden alle Durchläufe erfolgreich abgeschlossen, können Sie dies mit der Eigenschaft *IsCompleted* bestimmen:

```
If res.IsCompleted Then
    Console.WriteLine("Kein Abbruch")
Else
```

Falls nicht, können Sie auswerten wann der Abbruch erfolgte:

```
    Console.WriteLine("Abbruch bei " + res.LowestBreakIteration.Value.ToString())
End If
```

Wird über *IsCompleted False* zurückgegeben und gibt *LowestBreakIteration* den Wer *Nothing* 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 im Rahmen dieses Buchs jedoch 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]:

[1] Beachten Sie jedoch, dass in unserem Fall etwas Prozessorzeit für die Division "verschwendet" wird, dies aber sollte nicht den Vorteil der parallelen Verarbeitung zunichte machen.

BEISPIEL 10.9: Iteration mit einer Schrittweite von 0,001 zwischen 0 und 1 realisieren

VB
```
Imports System.Threading
Imports System.Threading.Tasks
...
        Parallel.For(0, 1000, Sub(i)
                                  Dim d As Double = i / 1000.0
                                  MeineMethode(d)
                              End Sub)
```

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) (source As IEnumerable(Of TSource),
                              body As Action(Of TSource) )
```

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

VB
```
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
        Dim Vornamen() As String = {"anton", "berta", "cäsar",
                                    "dora", "emil", "friedrich"}
```

Der Aufruf gestaltet sich einfacher als bei der *Parallel.For*-Lösung:

```
        Parallel.ForEach(Vornamen, Sub(vn) Console.WriteLine(vn.ToUpper()))
        Console.ReadKey()
    End Sub
...
```

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[1] 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 bieten sich zwei Varianten an:

- Zuweisen eines benannten Delegates,
- Verwendung eines Lambda-Ausdrucks, dieser kann wiederum benannte Methoden aufrufen.

BEISPIEL 10.11: Die drei Möglichkeiten für die Zuweisung des Task-Codes

VB

```
Imports System.Threading
Imports System.Threading.Tasks
...
```

Direkter Methodenaufruf:

```
    Sub test()
        Console.WriteLine("Methodenaufruf !")
    End Sub
...
    Sub Main()
        Dim mytask = New Task(AddressOf test)
        mytask.Start()
...
```

Verwendung eines Lambda-Ausdrucks:

```
        Dim mytask1 = New Task(Sub() Console.WriteLine("Anonyme Methode !"))
        mytask1.Start()
```

Für welche der beiden Varianten Sie sich entscheiden, hängt neben den eigenen Vorlieben sicher auch vom Umfang der Aufgabe ab.

[1] Hier ist mit "Task" die Aufgabe und nicht der Task aus dem Task-Manager gemeint.

10.5.2 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

VB

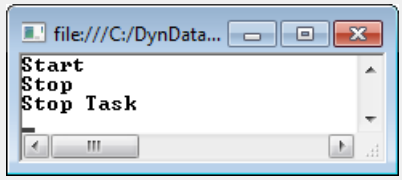
```
...
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
        Console.WriteLine("Start")
```

Ab hier läuft bereits der Task, den Rückgabewert der Methode *StartNew* können Sie zum Steuern und Abfragen des Task verwenden:

```
        Dim t = Task.Factory.StartNew(Sub()
                                          Thread.Sleep(1000)
                                          Console.WriteLine("Stop Task")
                                      End Sub
           )
        Console.WriteLine("Stop")
        Console.Read()
    End Sub
...
```

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*

VB

```
        Dim mytask = Task.Factory.StartNew(AddressOf MeineMethode,
                                           TaskCreationOptions.LongRunning)
```

BEISPIEL 10.14: Variante 2 explizites Starten

VB

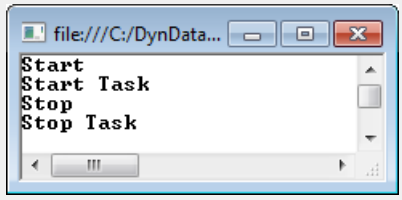
```
Imports System.Threading
Imports System.Threading.Tasks
...
```

BEISPIEL 10.14: Variante 2 explizites Starten

```
    Sub Main()
        Console.WriteLine("Start")
        Dim t = New Task(Sub()
                             Thread.Sleep(1000)
                             Console.WriteLine("Stop Task")
                         End Sub)
        Console.WriteLine("Start Task")
        t.Start()
        Console.WriteLine("Stop")
        Console.Read()
    End Sub
...
```

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*

VB

```
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
```

Instanz der Klasse definieren (Klassendefinition siehe am Ende des Listings):

```
        Dim kontakt As CKontakte = New CKontakte With {.Nachname = "Müller",
                                   .Vorname = "Heinz", .Telefon = "01901234567"}
        Console.WriteLine("Start")
```

Hier definieren wir einen Eingabeparameter für die Lambda-Funktion:

```
        Dim t = New Task(Sub(myData)
```

BEISPIEL 10.15: Übergabe eines Objekts vom Typ *CKontakte*

Wir müssen den Übergabewert typisieren:

```
Dim myLocalObject As CKontakte = CType(myData, CKontakte)
Thread.Sleep(1000)
```

Verwenden der Daten:

```
Console.WriteLine(myLocalObject.Nachname)
Console.WriteLine(myLocalObject.Vorname)
Console.WriteLine(myLocalObject.Telefon)
Console.WriteLine("Stop Task")
```

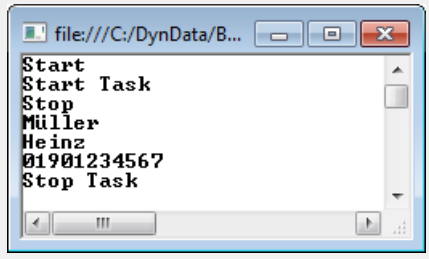
Hier findet die Übergabe der Daten an den Task statt:

```
                    End Sub, kontakt)
        Console.WriteLine("Start Task")
        t.Start()
        Console.WriteLine("Stop")
        Console.Read()
    End Sub
```

Die Klassendefinition:

```
Public Class CKontakte
    Public Nachname As String
    Public Vorname As String
    Public Telefon As String
End Class
```

Ergebnis



HINWEIS: Möchten Sie Daten aus dem Task zurückgeben, sollten Sie sich noch etwas gedulden, ab Seite 604 finden Sie die Lösung.

10.5.4 Wie warte ich auf das Taskende?

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
- 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 naheliegende Lösung bietet die *Wait*-Methode, die Sie über die aktuelle *Task*-Instanz aufrufen können.

BEISPIEL 10.16: Warten auf das Task-Ende

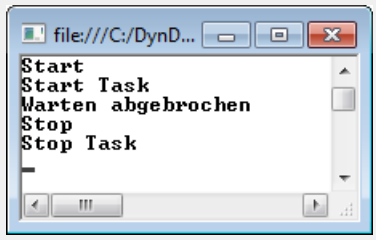
VB

```
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
        Console.WriteLine("Start")
        Dim t = New Task(Sub()
                             Thread.Sleep(5000)
                             Console.WriteLine("Stop Task")
                         End Sub)
        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) Then
            Console.WriteLine("Warten erfolgreich")
        Else
            Console.WriteLine("Warten abgebrochen")
        End If
        Console.WriteLine("Stop")
        Console.ReadKey()
    End Sub
...
```

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*

VB
```
...
    Dim t1 = Task.Factory.StartNew(Sub() Thread.Sleep(3000))
    Dim t2 = Task.Factory.StartNew(Sub() Thread.Sleep(5000))
    Dim t3 = Task.Factory.StartNew(Sub() 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

VB
```
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
```

Datentyp für die Rückgabe definieren:

```
        Dim t As New Task(Of Integer)(Function()
                                          Thread.Sleep(5000)
```

Hier erfolgt das Zuweisen des Rückgabewertes:

```
                                          Return 27
                                      End Function)
       Console.WriteLine("Start Task")
```

Task starten:

```
       t.Start()
```

BEISPIEL 10.18: Rückgabe eines Integerwertes

Ergebnis auswerten:

```
        Console.WriteLine("Rückgabewert = " & t.Result.ToString())
    End Sub
...
```

HINWEIS: Der eine oder andere 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)

VB

```
...
Imports System.IO
Imports System.Threading
Imports System.Threading.Tasks
...
    Sub Main()
```

Hier wir der Task definiert:

```
        Dim t1 As New Task(Of FileInfo())(Function()
                                              Dim di As New DirectoryInfo("c:\")
                                              Return di.GetFiles()
                                          End Function)
```

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()
```

BEISPIEL 10.19: Task mit Rückgabewert (Collection)

Rückgabewerte abrufen (hier verläuft die Verarbeitung synchron):

```
        Console.WriteLine("Rückgabewerte")
        For Each f As FileInfo In t1.Result
            Console.WriteLine(f.FullName)
        Next f
        Console.ReadKey()
...
```

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 *Task.Factory.StartNew*-Methode wählen.

VB

```
...
        Dim t1 = Task.Factory.StartNew(Function()
                                           Dim di As New DirectoryInfo("c:\")
                                           Return di.GetFiles()
                                       End Function)
        Console.WriteLine("Rückgabewerte")
        For Each f As FileInfo In t1.Result
            Console.WriteLine(f.FullName)
        Next f
...
```

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

VB

Indem Sie neue Objekte erzeugen, ist es problemlos möglich, jede Art von Informationen aus den Tasks zurückzugeben:

```
        Dim t2 = Task.Factory.StartNew(Function() New With {.Zeit = Date.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")
```

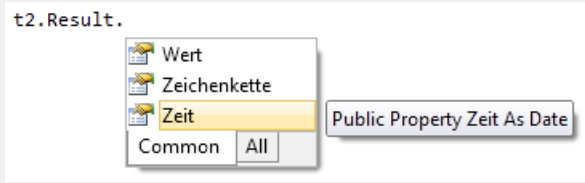
BEISPIEL 10.21: Mehrere (drei) Rückgabewerte definieren

Rückgabewerte anzeigen:

```
Console.WriteLine(t2.Result.Wert)
Console.WriteLine(t2.Result.Zeichenkette)
Console.WriteLine(t2.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.

[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)

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Insert(0, "Task starten ...")
```

Task definieren und starten:

```
        Dim myTask = Task.Factory.StartNew(Function()
                                               Dim i As Integer = 0
                                               Do
```

Hier steigen wir später aus:

```
                                                   If i = 10 Then Return 0
```

Ein paar Lebenszeichen an das Consolenfenster senden:

```
                                                   Debug.WriteLine("Wert von i = " &
                                                                   i.ToString())
                                                   Thread.Sleep(500)
                                                   i += 1
                                               Loop
                                           End Function)
```

Hier warten wir auf den Task[2], 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)
    End Sub
```

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 neuen Standard-Lösung für das Abbrechen von Tasks (und parallelen Threads) angelangt. Wie das folgende Beispiel zeigt, handelt es sich nicht um eine

[2] Eigentlich recht sinnlos, da dadurch die Oberfläche blockiert wird.

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 zum einen im Rahmen eines *Wait*-Statements oder bei der Auswertung eines Task-Rückgabewertes erfolgen (siehe folgendes Beispiel).

BEISPIEL 10.23: Task per Exception abbrechen

VB

```
Imports System.Threading
Imports System.Threading.Tasks
Public Class Form1
```

Zunächst brauchen wir eine zentrale *CancellationTokenSource* und einen *CancellationToken*:

```
    Private cts As CancellationTokenSource
    Private token As CancellationToken
    Private myTask As Task

    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

CancellationTokenSource und *CancellationToken* initialisieren:

```
        cts = New CancellationTokenSource()
        token = cts.Token
    End Sub
```

Hier kommen wir zum Starten des eigentlichen Tasks:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Insert(0, "Task starten ...")
        myTask = Task.Factory.StartNew(Sub()
                                           Dim i As Integer = 0
                                           Do
```

Etwas Aktivität im Ausgabefenster realisieren:

```
                                             Debug.WriteLine("Wert von i = " &
                                                             i.ToString())
```

BEISPIEL 10.23: Task per Exception abbrechen

Hier reagieren wir auf eine Abbruchanforderung:

```
                                    If token.IsCancellationRequested Then
                                     ' Hier noch sinnvolles Beenden möglich
```

Hier wir eine Exception ausgelöst, der Task endet damit:

```
                                        token.ThrowIfCancellationRequested()
                                    End If
                                    Thread.Sleep(500)
                                    i += 1
                                Loop
```

Wichtig: Der Token muss an den Task beim Erstellen übergeben werden:

```
                                End Sub, token)
    End Sub
```

Die Routine zum Abbruch des Task:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        ListBox1.Items.Insert(0, "Abbruch anfordern ...")
```

Hier wird die Aufforderung ausgelöst:

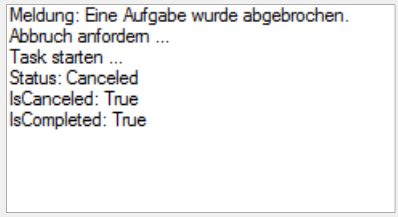
```
        cts.Cancel()
```

Im Rahmen eines *Wait*-Statements fangen wir den Fehler auf und zeigen den Fehler an:

```
        Try
            myTask.Wait()
        Catch ex As AggregateException
            For Each v In ex.InnerExceptions
                ListBox1.Items.Insert(0, "Meldung: " & v.Message)
            Next v
        End Try
        ListBox1.Items.Add("Status: " & myTask.Status.ToString())
        ListBox1.Items.Add("IsCanceled: " & myTask.IsCanceled)
        ListBox1.Items.Add("IsCompleted: " & myTask.IsCompleted)
    End Sub
```

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 *For Each*-Schleife die *InnerExceptions* aus.

BEISPIEL 10.24: (Auszug aus dem vorhergehenden Beispiel)

VB
```
Try
    myTask.Wait()
Catch ex As AggregateException
    For Each v In ex.InnerExceptions
        ListBox1.Items.Insert(0, "Meldung: " & v.Message)
    Next v
End Try
```

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.

[1] Ein Umstand, der sicher viele gute, aber leider auch unlesbare Programme hervorbringen wird ...

Mehr zu diese Thematik finden Sie im Abschnitt 11.2, wo wir uns speziell dem Debugging widmen wollen.

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 Userinterface

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

1. auf das Ende des Tasks reagieren können,
2. 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

VB

Abrufen aller Dateien aus dem Verzeichnis[1] "C:\Windows\System32" in einem Hintergrund-Task und Anzeige in einer *ListBox*, wenn alle Dateien abgerufen wurden.

```
...
Imports System.IO
Imports System.Threading
Imports System.Threading.Tasks
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Wir ermitteln den Task-Scheduler des UI-Threads:

```
        Dim uiSched = TaskScheduler.FromCurrentSynchronizationContext()
```

Neuen Task definieren und starten:

```
        Task.Factory.StartNew(Function()
                                  Dim di As New DirectoryInfo("c:\windows\system32")
                                  Thread.Sleep(3000)
```

[1] Eventuell anpassen ...

BEISPIEL 10.25: Warten auf Task-Ende

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):

```
                            End Function).ContinueWith(Sub(t)
                                  ListBox1.Items.Clear()
```

Task-Rückgabewert auswerten:

```
                                  For Each f As FileInfo In t.Result
                                     ListBox1.Items.Add(f.FullName)
                                  Next f
```

Achtung: Wir müssen noch den passenden Task-Scheduler angeben:

```
                           End Sub, uiSched)
    End Sub
End Class
```

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.

VB

```
Imports System.Threading
Imports System.Threading.Tasks
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Zunächst ermitteln wir wieder das *TaskScheduler*-Objekt für die Oberfläche:

```
        Dim 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):

BEISPIEL 10.26: Aktualisieren einer *ListBox* aus einem laufenden Task heraus.

```
        Dim token = Task.Factory.CancellationToken
```

Hier starten wir unseren Arbeits-Task:

```
        Task.Factory.StartNew(Sub()
                                  For i As Integer = 0 To 9
```

Wir wollen die Oberfläche bei jedem Schleifendurchlauf aktualisieren und erzeugen dazu einen neuen Anzeige-Task, der im Context der Oberfläche läuft:

```
                                      Task.Factory.StartNew(Sub() ListBox1.Items.Insert(0,
                                       "Wert von i = " & i.ToString()), token,
                                      TaskCreationOptions.None, uiSched)
```

Hier legen wir eine Verschnaufpause im Arbeits-Task ein:

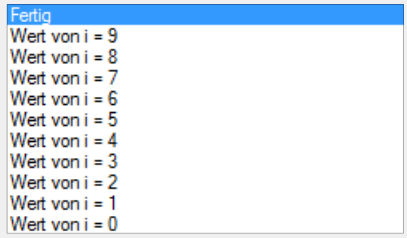
```
                                      Thread.Sleep(1000)
                                  Next i
```

Ist der Arbeits-Task beendet, zeigen wir noch eine Abschlussmeldung an:

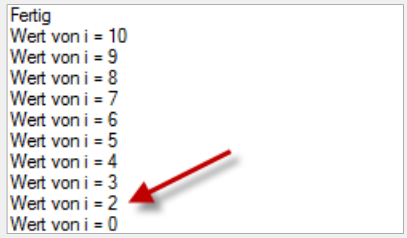
```
                              End Sub).ContinueWith(Sub(t) _
                                 ListBox1.Items.Insert(0, "Fertig"), uiSched)
    End Sub
End Class
```

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 Schleifen-

BEISPIEL 10.26: Aktualisieren einer *ListBox* aus einem laufenden Task heraus.

variable i nicht "threadsicher", die Anzeige erfolgt mal mit dem alten und mal mit dem neuen Schleifenwert, 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(Sub()
                                  For i As Integer = 0 To 9
                                      Thread.Sleep(1000)
                                      Dim t = Task.Factory.StartNew(Sub() _
                                              ListBox1.Items.Insert(0, "Wert von i = " &
                                              i.ToString()), token,
                                              TaskCreationOptions.None, uiSched)
                                      t.Wait()
```

Jetzt wartet der Arbeits-Thread auf den Anzeige-Thread!

```
                                  Next i
                              End Sub).ContinueWith(Sub(t) _
                                                    ListBox1.Items.Insert(0,
                                                    "Fertig"), uiSched)
...
```

Mit den Lösungen aus den beiden vorhergehenden Abschnitten dürften zwei Herzenswünsche des Task-Programmierers in Erfüllung gehen, vermutlich werden diese Lösungen aber auch dazu beitragen, dass einige Entwickler wieder etwas entnervt auf den Bildschirm starren und Ihre eigenen Programme nicht mehr verstehen.

10.7 Weitere Datenstrukturen im Überblick

Nachdem wir uns durch den Hauptteil der neuen TPL gewühlt haben, wollen wir abschließend noch einen kurzen Blick auf die neuen 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 Umfangsgründen nicht möglich, dieses Thema in aller 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 nötig, müssen die entsprechenden Anweisungen akribisch gesichert und damit threadsicher gemacht werden (z.B. mit *Synclock*). Um dem Programmierer dieses Arbeit abzunehmen und gleichzeitig 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 6.

10.7.2 Primitive für die Threadsynchronisation

Wie schon im Abschnitt 9.3 ab Seite 523 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 Neuerungen, 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 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 wollen wir an dieser Stelle verzichten, dies wurde alles beibehalten, lediglich die Logik hinter den Kulissen musste teilweise komplett umgebaut werden.

10.9.1 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.

10.9.2 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.

```
Imports System.Threading
Imports System.Threading.Tasks

Friend NotInheritable Class myGlobal
```

Die Instanzen unserer obigen Klassen:

```
    Public Shared Controller As cController
    Public Shared Schiff As Schiff
    Public Shared LKWs(2) As cLKW
```

Den Zugriff auf die Oberfläche ermöglicht uns die folgende Variable, in der wir die Instanz von *Form1* speichern werden:

```
Public Shared myForm As Form1
```

Den *TaskScheduler* des UI speichern wir ebenfalls, so können wir aus den Tasks heraus auf die Oberfläche zugreifen, ohne explizit *Invoke* zu verwenden:

```
Public Shared uiSched As TaskScheduler
```

Für die Vollständigkeit aller Parameter beim Erstellen einiger Tasks benötigen wir noch einen *CancellationToken*:

```
Private Shared token As CancellationToken = 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 Shared Sub Anzeige(msg As String)
```

Task erzeugen und starten:

```
        Task.Factory.StartNew(Sub()
                                  If msg <> "" Then
                                      myForm.ListBox1.Items.Insert(0, Date.Now & " : " & msg)
                                  Else
                                      myForm.Label3.Text = myGlobal.Schiff.Ladung.ToString()
                                      myForm.Label4.Text =
                                          myGlobal.Schiff.Transportiert.ToString()
                                      myForm.Label6.Text =
                                          myGlobal.Controller.Lagerbestand.ToString()
                                      myForm.Label10.Text = myGlobal.LKWs(0).kmStand.ToString()
                                      myForm.Label11.Text = myGlobal.LKWs(1).kmStand.ToString()
                                      myForm.Label12.Text = myGlobal.LKWs(2).kmStand.ToString()
                                      myForm.PictureBox2.Image = myGlobal.LKWs(0).Image
                                      myForm.PictureBox3.Image = myGlobal.LKWs(1).Image
                                      myForm.PictureBox4.Image = myGlobal.LKWs(2).Image
                                      myForm.PictureBox1.Left = myGlobal.Schiff.pos
                                      myForm.PictureBox2.Left = myGlobal.LKWs(0).Pos
                                      myForm.PictureBox3.Left = myGlobal.LKWs(1).Pos
                                      myForm.PictureBox4.Left = myGlobal.LKWs(2).Pos
                                  End If
                              End Sub, token, TaskCreationOptions.None, uiSched)
    End Sub
End Class
```

10.9.3 Controller

Für die übergreifende Steuerung sorgt unser Controller, den wir ebenfalls "runderneuert" haben:

```
Imports System.Threading
Imports System.Threading.Tasks
```

```
Class cController
```

Der interne Lagerbestand, bevor die Ware von den LKWs abgeholt wird:

```
    Public Lagerbestand As Integer
```

Um keine CPU-Zeit zu verschwenden, warten wir mit einem *AutoResetEvent* bis neue Ware ins Lager gestellt wird:

```
    Private NeueWareDa As 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 Sub NeueWare()
        Lagerbestand += 1
```

Hier geben wir die Sperre frei, der Controller kann wieder arbeiten:

```
        NeueWareDa.Set()
    End Sub
```

Unser "Hauptprogramm":

```
    Public Sub New()
```

Task mit Endlosschleife erzeugen:

```
        Task.Factory.StartNew(Sub()
                                  myGlobal.Anzeige("")
                                  myGlobal.Anzeige("Controller: Gestartet")
                                  Dim i As Integer = 0
                                  Do
```

Ist der Lagerbestand größer null, wird versucht, die LKWs zu beladen:

```
                                      Do While Lagerbestand > 0
                                          If myGlobal.LKWs(i).ImLager Then
                                              myGlobal.Anzeige("Controller: Lkw beladen")
                                              myGlobal.LKWs(i).Abtransport()
                                          End If
```

Falls der gewünschte LKW nicht vorhanden ist, versuchen wir es mit dem nächsten, etwas Pause zwischendurch verhindert unsinnige CPU-Belastung[1]:

```
                                          i += 1
                                          If i > 2 Then i = 0
                                          Thread.Sleep(100)
                                      Loop
```

[1] Wie im echten Leben ...

Falls keine Ware da ist gehen wir hier in den Dämmerschlaf über, erst wenn neue Ware hereinkommt, wird diese Sperre wieder freigegeben:

```
                                    NeueWareDa.WaitOne()
                                Loop
                            End Sub)
    End Sub
End Class
```

10.9.4 LKWs

```
Imports System.Threading
Imports System.Threading.Tasks

Friend Class cLKW
    Public Nr As Integer
    Public Pos As Integer
    Public kmStand As Integer
    Public ImLager As Boolean
    Private Img1 As Image
    Private Img2 As Image
    Public Image As Image
```

LKW initialisieren:

```
    Public Sub New(nr As Integer)
        Me.Nr = nr
        kmStand = 0
        Pos = 280
        ImLager = True
        Img1 = My.Resources.Auto1
        Img2 = My.Resources.Auto2
        Image = Img1
    End Sub
```

Diese Methode wird durch den Controller aufgerufen und gewährleistet sowohl Hinfahrt/Rückfahrt, als auch die reibungslose Übergabe der Ware an das Schiff:

```
    Public Sub Abtransport()
        ImLager = False
```

Den Lagerbestand verringern:

```
        myGlobal.Controller.Lagerbestand -= 1
```

Neuer Task:

```
        Task.Factory.StartNew(Sub()
                                  Thread.Sleep(200)
                                  myGlobal.Anzeige("Lkw_" & Nr.ToString() & ": Fahrtbeginn")
                                  Image = Img1
```

Hinfahrt:

```
                                For Pos = 280 To 1 Step -2
                                    Thread.Sleep(30)
                                    kmStand += 1
                                    myGlobal.Anzeige("")
                                Next Pos
```

Ankunft im Hafen, auf Schiff warten:

```
                                myGlobal.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 To 279 Step 2
                                    Thread.Sleep(20)
                                    kmStand += 1
                                    myGlobal.Anzeige("")
                                Next Pos
                                ImLager = True
                                myGlobal.Anzeige("Lkw_" & Nr.ToString() & ": Fahrtende")
                            End Sub)
      End Sub

End Class
```

10.9.5 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:

```
Imports System.Threading
Imports System.Threading.Tasks

 Friend Class Schiff
    Public Ladung As Integer = 0
    Public Transportiert As Integer = 0
    Public pos As Integer = 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 ImHafen As New ManualResetEvent(True)
    Private NeueWareDa As New AutoResetEvent(False)
```

Die zentrale Transport-Routine:

```
    Public Sub Transport()
```

Die Sperre manuell setzen:

```
        ImHafen.Reset()
```

Abfahrt:

```
        myGlobal.Anzeige("Schiff: Abfahrt")
        For pos = 200 To 1 Step -2
            Thread.Sleep(100)
            myGlobal.Anzeige("")
        Next pos
```

Im Hafen die Ladung löschen:

```
        myGlobal.Anzeige("Schiff: Ladung löschen")
        Do While Ladung > 0
            Thread.Sleep(400)
```

Da nur noch ein *Task* auf *Ladung* bzw. *Transportiert* zugreifen kann, können wir jetzt auf einen *Monitor/SyncLock*-Abschnitt verzichten:

```
            Ladung -= 1
            Transportiert += 1
            myGlobal.Anzeige("")
        Loop
```

Die Rückfahrt:

```
        For pos = 0 To 199 Step 2
            Thread.Sleep(60)
            myGlobal.Anzeige("")
        Next pos
        myGlobal.Anzeige("Schiff: Warten ...")
```

Sperre manuell zurücksetzen:

```
        ImHafen.Set()
    End Sub
```

Die blockierende Methode *Beladen:*

```
    Public Sub Beladen()
```

Hier warten wir auf das Schiff:

```
        ImHafen.WaitOne()
        Ladung += 1
        myGlobal.Anzeige("Schiff: Beladen (" & Ladung.ToString() & ")")
```

Sperre für den Haupt-Task freigeben:

```
        NeueWareDa.Set()
    End Sub
```

Der Konstruktor:

```
    Public Sub New()
```

Neuen Task erstellen:

```
        Task.Factory.StartNew(Sub()
```

Eine Endlosschleife:

```
                                  Do
```

Wenn keine Ware geladen ist warten wir, bis neue Ladung per LKW heran transportiert wird (unbegrenzt):

```
                                      If Ladung = 0 Then
                                          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) Then
                                              If Ladung = 5 Then
                                                  Transport()
                                              End If
```

Andernfalls ist nur ein Timeout aufgetreten, wir fahren im Zweifel auch mit einem Stück weg:

```
                                          Else
                                              myGlobal.Anzeige("Schiff: Zeit abgelaufen")
                                              Transport()
                                          End If
                                      End If
                                  Loop
                              End Sub)
    End Sub
End Class
```

10.9.6 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:

```
Imports System.Threading
Imports System.Threading.Tasks
Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Unsere Objekte initialisieren:

```
        myGlobal.uiSched = TaskScheduler.FromCurrentSynchronizationContext()
        myGlobal.LKWs(0) = New cLKW(0)
        myGlobal.LKWs(1) = New cLKW(1)
        myGlobal.LKWs(2) = New cLKW(2)
        myGlobal.Schiff = New Schiff()
```

```
        myGlobal.Controller = New cController()
```

Achtung: Die Referenz für das Formular speichern:

```
        myGlobal.myForm = Me          ' Referenz für das Formular speichern
    End Sub
```

HINWEIS: Vergessen Sie in diesem Zusammenhang bitte nicht, dass die Oberflächen-Steuerelemente, die von der Klasse *myGlobal* 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 Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        myGlobal.Controller.NeueWare()
    End Sub
```

Das Schließen des Formulars gestaltet sich recht unkompliziert, das Beenden der Task wollen wir ohne extra Aufwand bewerkstelligen:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Close()
    End Sub
End Class
```

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 nur 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 Teil 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 widmet sich der zweite Teil des Kapitels.

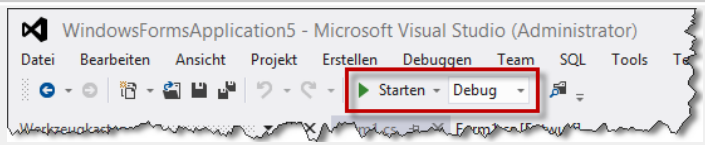
11.1 Der Debugger

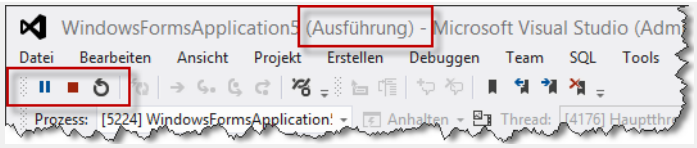
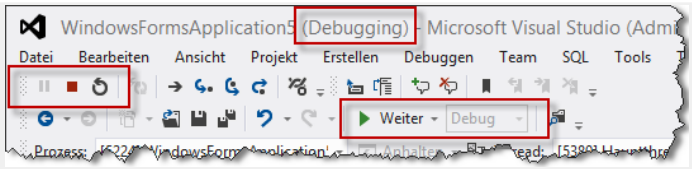
Der in Visual Studio 2015 integrierte Debugger ist ein recht komplexes Werkzeug, ausgestattet mit einer Fülle von Features und Fenstern. Zumindest den Umgang mit den elementaren Debug-Funktionen (auf diese wollen wir uns im Folgenden beschränken) sollte der Programmierer im Schlaf beherrschen, spart 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 können 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:

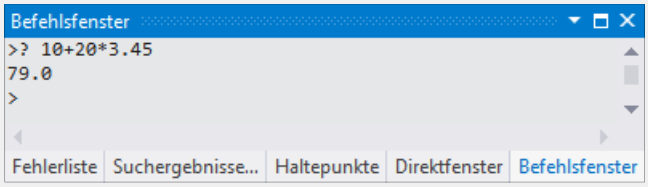
- Befehlsfenster (*Ansicht|Weitere Fenster|Befehlsfenster*)
- Direktfenster (*Debuggen|Fenster|Direkt*)
- Lokalfenster (*Debuggen|Fenster|Lokal*)
- Überwachungsfenster (*Debuggen|Fenster|Überwachen*)
- Autofenster (*Debuggen|Fenster|Auto*)
- Ausgabefenster (*Ansicht |Ausgabe* oder *Debuggen|Fenster| 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 VB-Anweisungen experimentieren.

BEISPIEL 11.1: Ausführen einer Rechenaufgabe (10+20*3,45=79,0) im unmittelbaren Modus



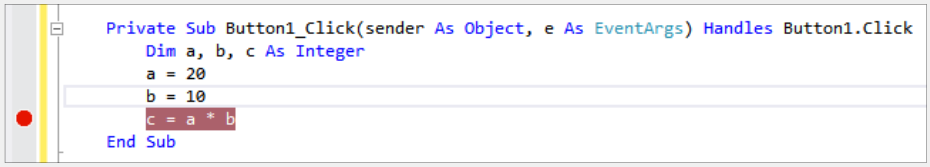
HINWEIS: Nützlich ist das Befehlsfenster unter anderem auch für das schnelle Ausprobieren diverser Optionen (zum Beispiel *Format*-Anweisungen o.ä.).

Direktfenster

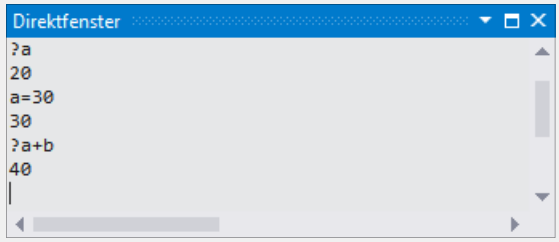
Das Direktfenster erlaubt es Ihnen, sich während des Debugging-Prozesses Werte von Variablen oder Ausdrücken ausgeben zu lassen, oder sogar diese Werte zu ändern.

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 so in den Unterbrechungsmodus:

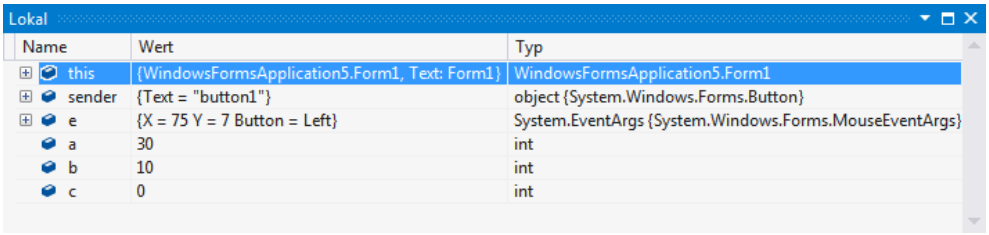


Im Direktfenster können Sie nun verschiedene Operationen durchführen:



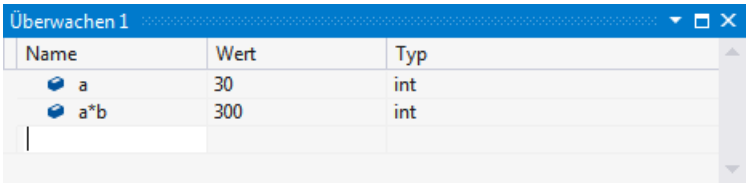
Lokal-Fenster

Hier lassen sich – vorausgesetzt das Programm befindet sich im Unterbrechungsmodus – Namen, Wert und Typ aller aktuellen Variablen und Objekte anzeigen:



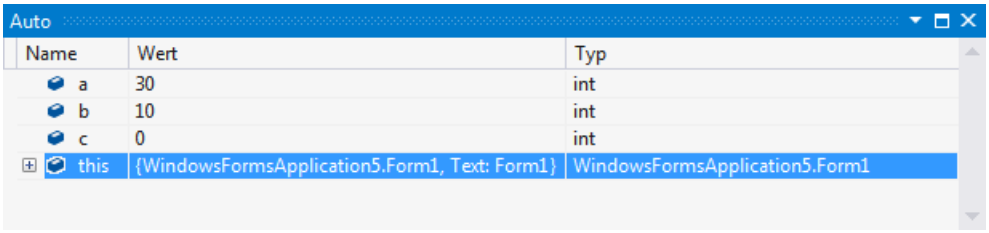
Überwachungsfenster

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).

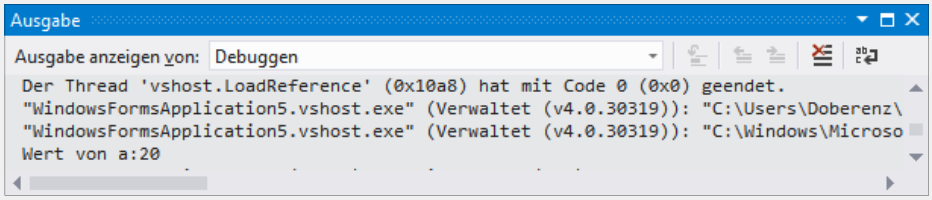
BEISPIEL 11.3: Anzeige von Werten im Ausgabefenster

VB
```
Imports System.Diagnostics
...
    Dim a,b As Integer
    a = 20
    Debug.WriteLine("Wert von a:" & a)
```

BEISPIEL 11.3: Anzeige von Werten im Ausgabefenster

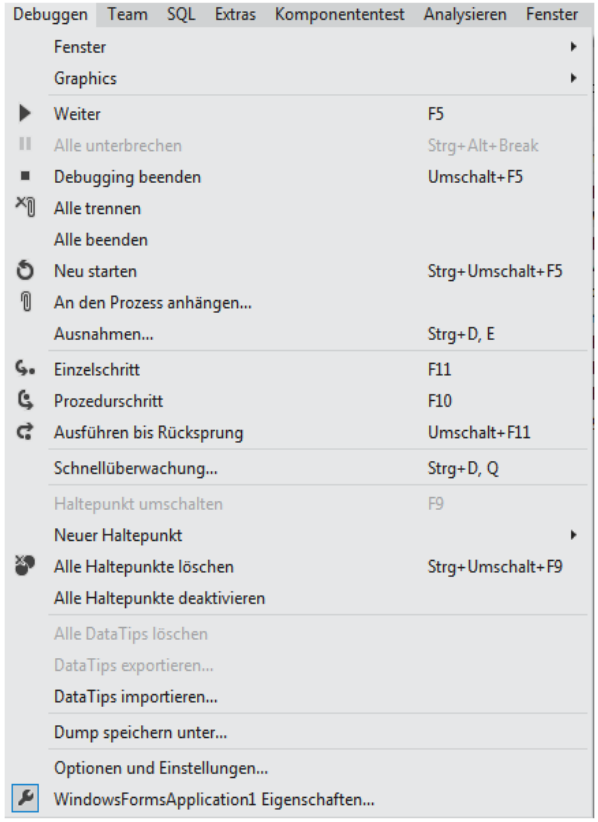
Ergebnis

Das dazugehörige Ausgabefenster:



11.1.3 Debugging-Optionen

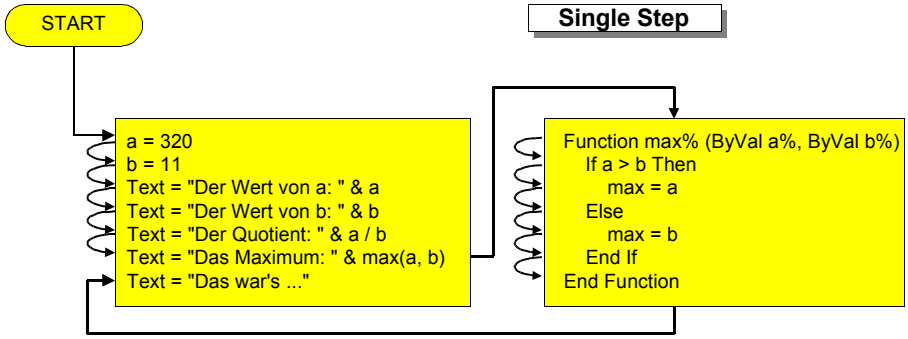
An der Anzahl der Einträge im Menü *Debuggen* erahnen Sie bereits die Vielfalt der Debugging-Möglichkeiten:



Viele Fehler lassen sich nur durch schrittweises Abarbeiten des Quelltextes eingrenzen, Sie können dabei zwischen mehreren Varianten wählen:

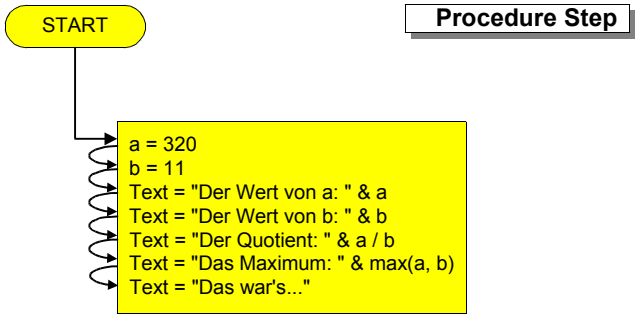
Einzelschritt-Modus

Im *Single-Step*-(Einzelschritt-)Modus durchlaufen Sie das Programm (mit *F11*) zeilenweise. Nach der Ausführung einer Anweisung liegt der Unterbrechungsmodus vor.



Prozedurschritt-Modus

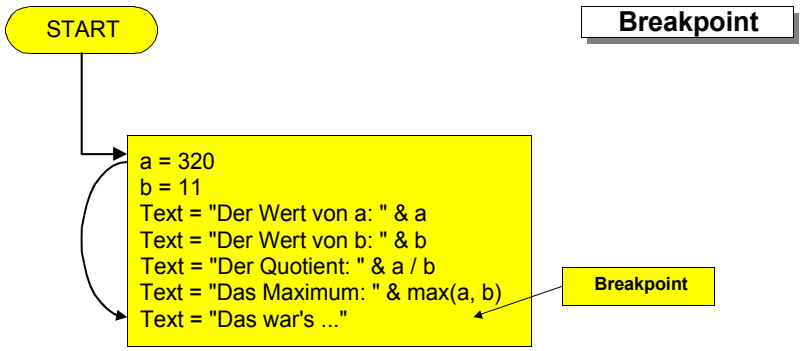
Bei umfangreicheren Programmen ist das zeilenweise Abarbeiten recht mühsam. Abhilfe schafft hier der *Procedure-Step*-(Prozedurschritt-)Modus (mit *F10*). Zwar wird nach wie vor jede einzelne Zeile verarbeitet, innerhalb von Unterprogrammen wird allerdings nicht mehr angehalten.



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 auszuprobieren, können Sie natürlich eine beliebige Anwendung nehmen. Die folgenden Ausführungen beziehen sich auf ein sehr einfaches Windows Forms-Programm, das sich an obige Abbildungen anlehnt und das lediglich aus einem *Form*ular besteht.

BEISPIEL 11.4: Einfache Debugging-Demo

VB

Nach dem Programmstart werden die folgenden Anweisungen ausgeführt:

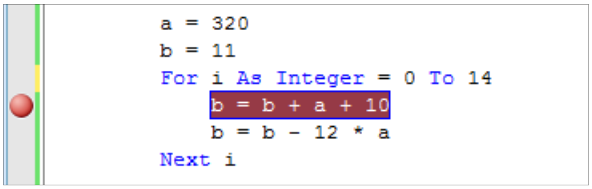
```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Dim a, b As Integer
        a = 320
        b = 11
        For i As Integer = 0 To 14
            b = b + a + 10
            b = b - 12 * a
        Next i
        ' hier wird später Code ergänzt!
 End Sub
```

HINWEIS: Das Programm hat, wie Sie sehen, keinerlei sinnvolle Funktion – hier geht es lediglich um die Erläuterung eines Prinzips.

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[1]. Genauso einfach lassen sich die Haltepunkte wieder entfernen – Sie brauchen nur darauf zu klicken.

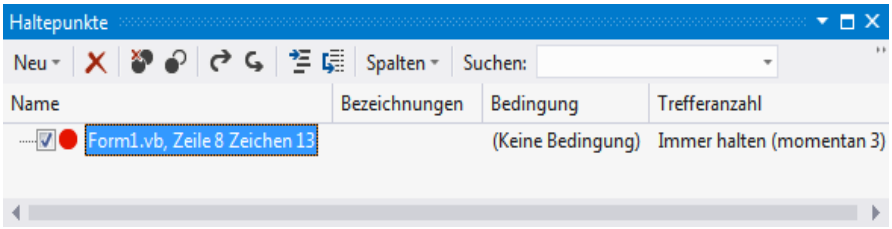
[1] Sie können den Breakpoint auch mit *F9* setzen.



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.

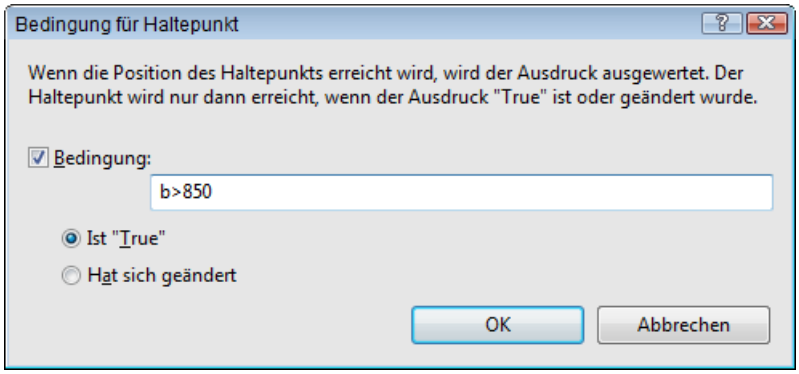
Besonders dann, wenn Sie mehrere Breakpoints gesetzt haben, sind Sie für eine Übersicht dankbar. Wählen Sie dazu das Menü *Debuggen|Fenster|Haltepunkte*. Es erscheint die Liste aller Haltepunkte. Wie Sie der folgenden Abbildung entnehmen, werden die Haltepunkte anhand ihrer Zeilennummer und des Moduls unterschieden.



HINWEIS: Um zu einem bestimmten Breakpoint 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:

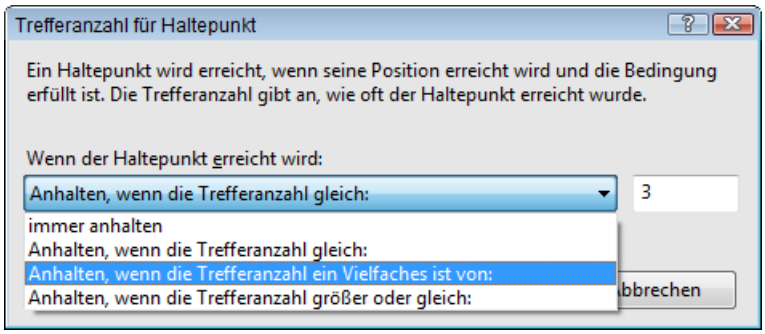


Wenn Sie jetzt unser Demoprogramm normal starten (*F5*), dürfte der Debugger erst dann die Programmausführung anhalten, wenn *c* einen Wert größer zwei hat (*b* ist zu diesem Zeitpunkt größer als *850*).

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:

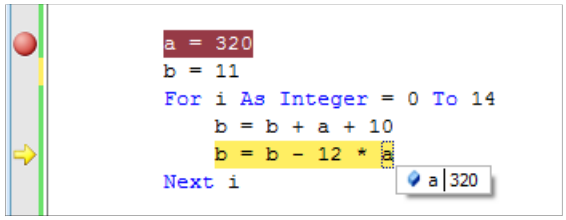


Nach obiger Einstellung und bei Beibehaltung unserer Abbruchbedingung stoppt der Debugger, wenn die Variable *b* den Wert fünf hat (zwei Durchläufe für die Abbruchbedingung und drei für die Trefferanzahl).

Einzelschritt-Modus

Haben Sie einen Breakpoint auf eine bestimmte Anweisung gesetzt, können Sie ab hier mit *F11* zeilenweise den Programmfortschritt beobachten (gelber Pfeil und gelbe Zeilenmarkierung).

Die Werte einzelner Variablen in der Ausführungsposition können Sie nun per QuickInfo kontrollieren, indem Sie einfach den Mauskursor darauf setzen:



Parallel dazu können Sie im Überwachungs- und im Lokalfenster die (rot eingefärbten) aktuellen Überwachungsausdrücke bzw. Variablenwerte verfolgen.

Für diese wie für alle anderen Betriebsarten des Debuggers gilt, dass Sie über *Shift+F5* (bzw. das Menü *Debuggen|Debuggen beenden*) den Debug-Modus verlassen können, um in den normalen Entwurfsmodus zurückzukehren.

Prozedurschritt

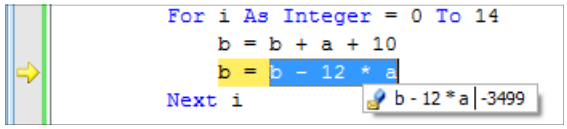
Bei dieser Variante verfahren Sie völlig analog zum Einzelschritt, nur dass Sie diesmal die *F10*-Taste (oder die Schaltfläche) verwenden. Sie werden in unserem Beispiel beobachten, dass bei den Anweisungen innerhalb der *Max*-Methode **nicht** angehalten wird. Den Prozedurschritt werden Sie also nur 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].und gehören zum (standardmäßig eingebundenen) Namespace *System.Diagnostics*.

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 620). Falls es nicht sichtbar ist, können Sie das Fenster über das Menü *Ansicht|Ausgabe* zur Anzeige bringen.

BEISPIEL 11.5: Ergänzung der Debugging-Demo von Seite 623

VB

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)
Dim f As Single = Convert.ToSingle(a / b)
```

[1] Auf den "feinen" Unterschied zwischen beiden Klassen werden wir erst an späterer Stelle eingehen.

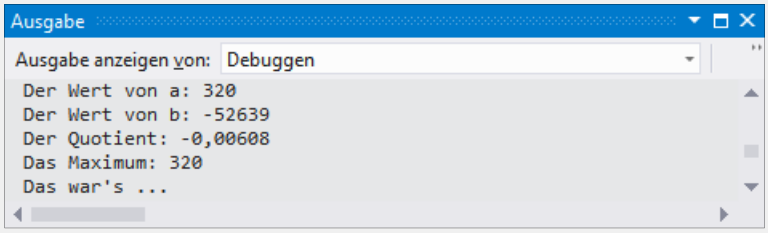
BEISPIEL 11.5: Ergänzung der Debugging-Demo von Seite 623

```
    Debug.WriteLine("Der Quotient: " + f.ToString("#0.00000"))
    Debug.WriteLine("Das Maximum: " & Max(a, b).ToString)
    Debug.WriteLine("Das war's ...")
```

Die dazu erforderliche Funktion *Max*:

```
Private Function Max(a As Integer, b As Integer) As Integer
    If b > a Then
        Return b
    Else
        Return a
    End If
End Function
```

Nach Programmstart finden sich im Ausgabefenster folgende Zeilen:



Wie Sie sehen ist – im "feinen" 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.

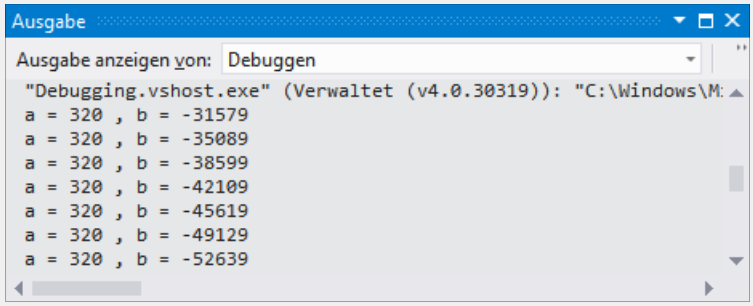
BEISPIEL 11.6: Ausgabe erfolgt nur, wenn die Schleifenvariable c größer gleich 10 ist

VB

Fügen Sie an das Ende der *For*-Schleife im Vorgängerbeispiel folgende Zeile ein:

```
Debug.WriteLineIf(b < -30000, "a = " & a.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

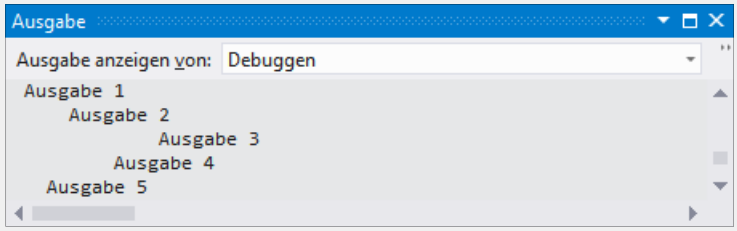
Das folgende Beispiel soll für Klarheit sorgen:

BEISPIEL 11.7: Demonstration verschiedener Texteinzüge

VB

```
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 der Annahme, dass a größer als 10000 sei

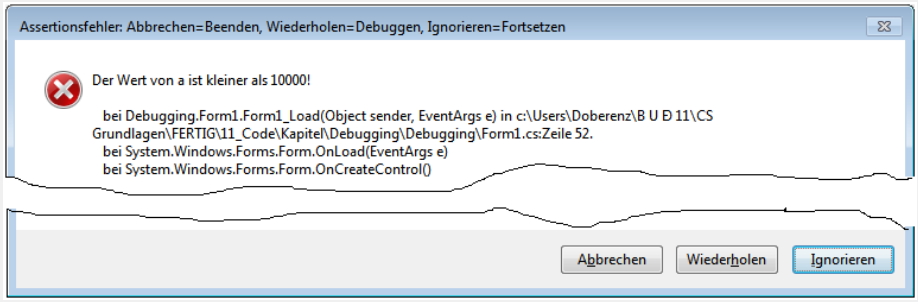
VB

```
Dim a As Integer = 5500
Debug.Assert(a > 10000, "Der Wert von a ist kleiner als 10000!")
```

Ergebnis

Nach dem Programmstart erscheint ein monströses Meldungsfenster, das hier nur in stark reduzierter Form dargestellt werden soll:

BEISPIEL 11.8: Meldung bei Verletzung der Annahme, dass a größer als 10000 sei



Fail

Diese Methode funktioniert ähnlich wie *Assert* und bietet eine elegante Alternative zum Aufruf von *MessageBox.Show*.

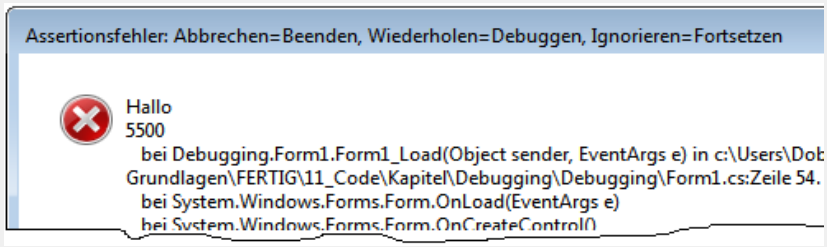
BEISPIEL 11.9:

VB

```
Dim a As Integer = 5500
Debug.Fail("Hallo", a.ToString)
```

Ergebnis

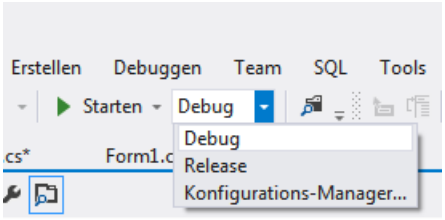
Das Meldungsfenster entspricht prinzipiell dem beim Aufruf der *Assert*-Methode (siehe obiges Beispiel).



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

VB
```
Dim myListener As 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 VB-Quellcode (siehe obiges Beispiel), sondern direkt in der *Anwendungskonfigurationsdatei* definieren.

Falls für Ihr Projekt eine *App.config* noch nicht existiert, können Sie eine solche über das Menü *Projekt/Neues Element hinzufügen...* erstellen lassen. Unter dem Knoten *Gemeinsame Elemente*/*Allgemein* wählen Sie die Vorlage *Anwendungskonfigurationsdatei.* Im Projektmappen-Explorer müsste jetzt eine *App.config* zu sehen sein.

Zwischen beiden *system.diagnostics*-Tags fügen Sie dann den im Folgenden fett hervorgehobenen XML-Code ein:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.diagnostics>
        <sources>
           <source name="DefaultSource" switchName="DefaultSwitch">
                <listeners>
                    <add name="FileLog"/>
                </listeners>
            </source>
        </sources>
...
      <trace autoflush="true">
        <listeners>
          <add name="debugListener"
           type="System.Diagnostics.TextWriterTraceListener"
           initializeData="DebugProtokoll.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.

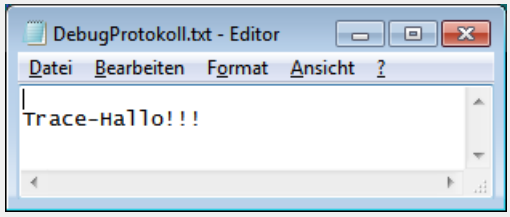
BEISPIEL 11.11: Test der obigen App.config

VB
```
Trace.WriteLine("Trace-Hallo!!!")
```

BEISPIEL 11.11: Test der obigen App.config

Ergebnis

Öffnen Sie die im *\bin\Debug*-Unterverzeichnis angelegte Datei mit einem Texteditor:



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 simple Textdatei zu schnöde ist, der kann die Ausgaben von *Debug*- und *Trace*-Objekt auch direkt in die Ereignisanzeige von Windows schreiben lassen.

BEISPIEL 11.12: *Trace*-Ausgaben in das Ereignisprotokoll schreiben

VB

Das gewünschte Protokoll aussuchen:

```
Dim log As New EventLog("Application")
```

Anwendungsname hinzufügen:

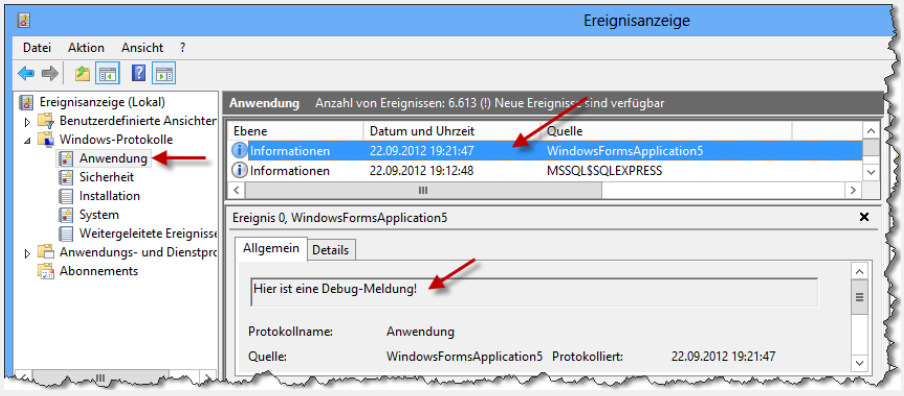
```
log.Source = Assembly.GetEntryAssembly().GetName().Name
```

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 mit Visual Basic 2012 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

VB

```
Imports System.Runtime.CompilerServices
Module Module1
```

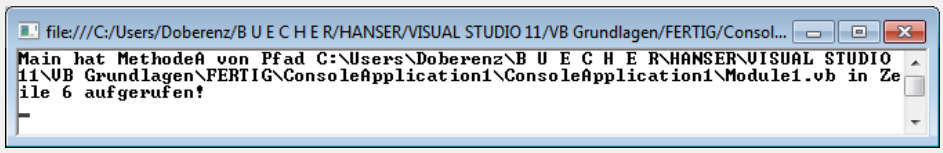
Main ist die aufrufende Methode (Caller):

```
    Sub Main()
        MethodeA()              ' Aufruf
        Console.ReadLine()
    End Sub
```

Die aufgerufene Methode:

```
    Sub MethodeA(<CallerMemberName> Optional callerName As String = "",
                 <CallerFilePath> Optional callerFilePath As String = "",
                 <CallerLineNumber> Optional callerLineNumber As Integer = 0)
        Console.WriteLine("{0} hat MethodeA von Pfad {1} in Zeile {2} aufgerufen!",
                          callerName, callerFilePath, callerLineNumber.ToString)
    End Sub
End Module
```

Ergebnis



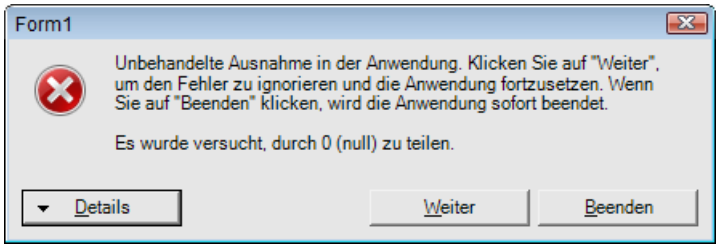
11.4 Fehlerbehandlung

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 Datenträger etc.
- zu wenig Speicher
- die gute alte 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:



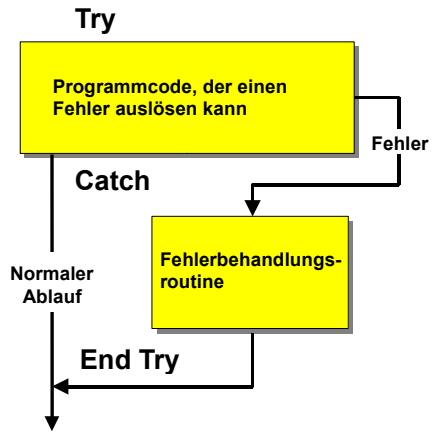
Sicherlich wollen Sie einen solchen Anblick nicht zum Markenzeichen Ihrer Programme werden lassen. Das haben Sie auch nicht nötig, denn Visual Basic bietet genügend Möglichkeiten für eine angemessene Fehlerbehandlung.

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":



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*

VB
```
Dim a, b, c As Integer
a = 10
b = 0
Try
    c = a / b
    Me.Text = c.ToString
Catch
    Me.Text = "Fehler"
End Try
```

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

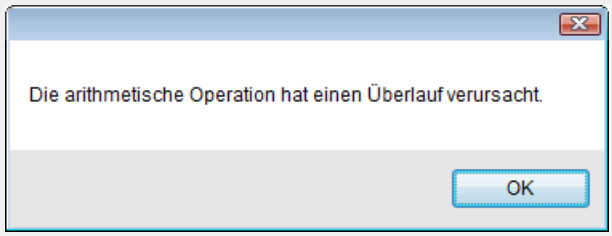
VB
```
Dim a, b, c As Integer
a = 10
b = 0
Try
    c = a / b
    Me.Text = c.ToString
```

BEISPIEL 11.15: "Gezielte" Fehlerbehandlung

```
    Catch mye As Exception
        MessageBox.Show(mye.Message)
        Me.Text = "Fehler"
    End Try
```

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 denn 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

VB

```
    Dim a, b, c As Integer
    a = 10
    b = 0
    Try
        c = a / b
        Me.Text = c.ToString
```

Spezielle(Fehlerklasse)

```
    Catch mye As System.OverflowException
        MessageBox.Show("Mal wieder eine Division durch null!")
```

Zur Sicherheit auch noch eine allgemeine Fehlerbehandlung:

```
    Catch mye As Exception
        MessageBox.Show(mye.Message)
        Me.Text = "Fehler"
    End Try
```

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

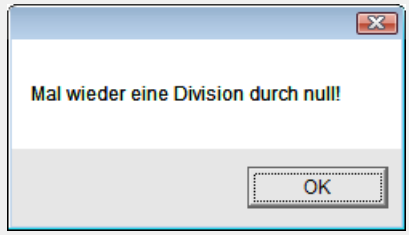
Bei den beiden oben genannten Varianten der Fehlerbehandlung wird die Programmausführung normal fortgesetzt. Möchten Sie jedoch, dass der Fehler zu einem Programmabbruch führt, können Sie diesen im *Catch*-Block erneut auslösen. Verwenden Sie dazu die *Throw*-Anweisung.

BEISPIEL 11.17: Fehler erneut auslösen

VB

```
Dim a, b, c As Integer
a = 10
b = 0
Try
    c = a / b
    Me.Text = c.ToString
Catch mye As System.OverflowException
    MessageBox.Show("Mal wieder eine Division durch null!")
    Throw mye
End Try
```

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

VB

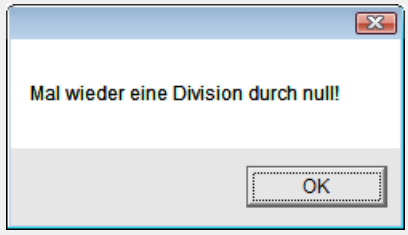
```
Private Sub Test()
    Dim a, b, c As Integer
    a = 10
    b = 0
    Try
```

BEISPIEL 11.18: Weiterreichen eines Fehlers an eine übergeordnete Fehlerbehandlung

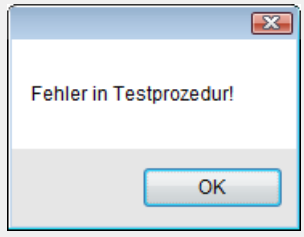
```
            c = a / b
            Me.Text = c.ToString
        Catch mye As System.OverflowException
            MessageBox.Show("Mal wieder eine Division durch null!")
            Throw mye
        End Try
        MessageBox.Show("Alles OK in Testprozedur!")
    End Sub

    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        Try
            Test()
        Catch
            MessageBox.Show("Fehler in Testprozedur!")
        End Try
        MessageBox.Show("Alles Ok im Hauptprogramm!")
    End Sub
```

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

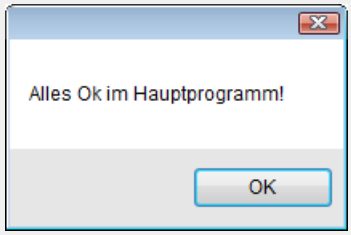


Die Anweisung *MessageBox.Show("Alles OK in Testprozedur!")* wird nie ausgeführt, da der erneut auftretende Fehler direkt in die übergeordnete Fehlerbehandlung, d h. bei *Button_Click*, verzweigt. Hier wird der Fehler endgültig abgearbeitet:



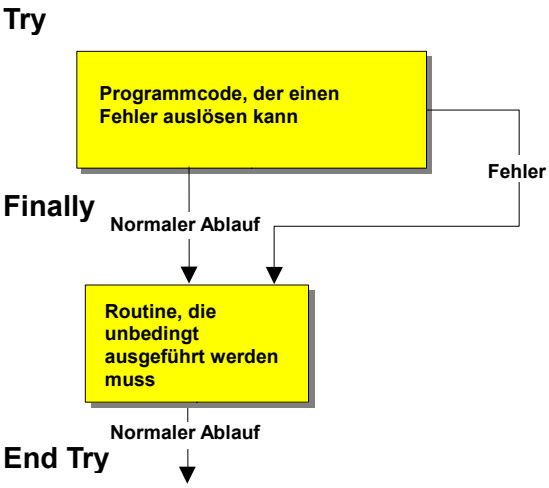
BEISPIEL 11.18: Weiterreichen eines Fehlers an eine übergeordnete Fehlerbehandlung

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, eine Access-Datenbank zu öffnen

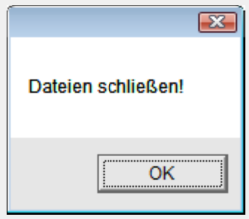
VB

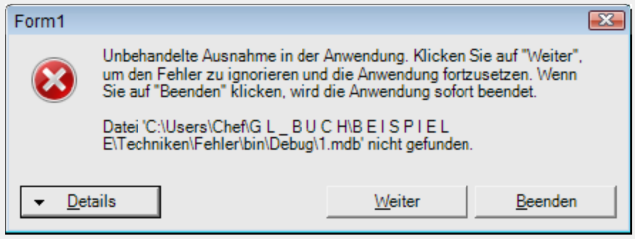
```
Imports System.Data.OleDb
...
    Dim conn1 As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Test.mdb")
    Try
        conn1.Open()
    Finally
        MessageBox.Show("Dateien schließen!")
        conn1.Close()
    End Try
```

Der Versuch, die Connection 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)

VB

```
    Dim conn1 As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0 Data Source=Test.mdb")
    Try
        conn1.Open()
```

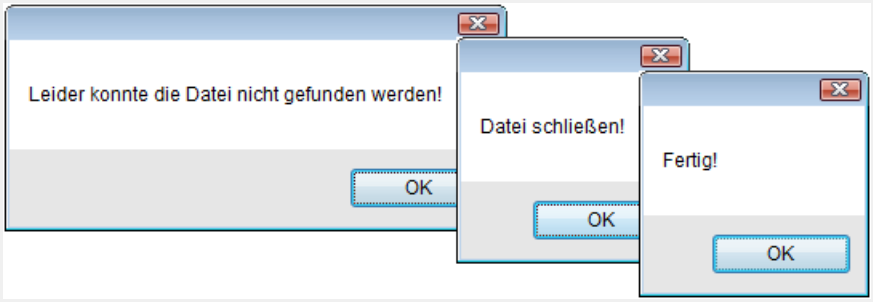
[1] In der Visual Studio-IDE erscheinen die Dialoge in einer anderen Reihenfolge, für uns ist jedoch das reine Laufzeitverhalten von Interesse.

BEISPIEL 11.20: *Try-Catch-Finally* (siehe Vorgängerbeispiel)

```
    Catch
        MessageBox.Show("Leider konnte die Datei nicht gefunden werden!")
    Finally
        MessageBox.Show("Datei schließen!")
        conn1.Close()
    End Try
    MessageBox.Show("Fertig!")
```

Ergebnis

Die Reihenfolge der Dialogboxen:



Wie Sie sehen, wird die Programmausführung anschließend normal fortgesetzt.

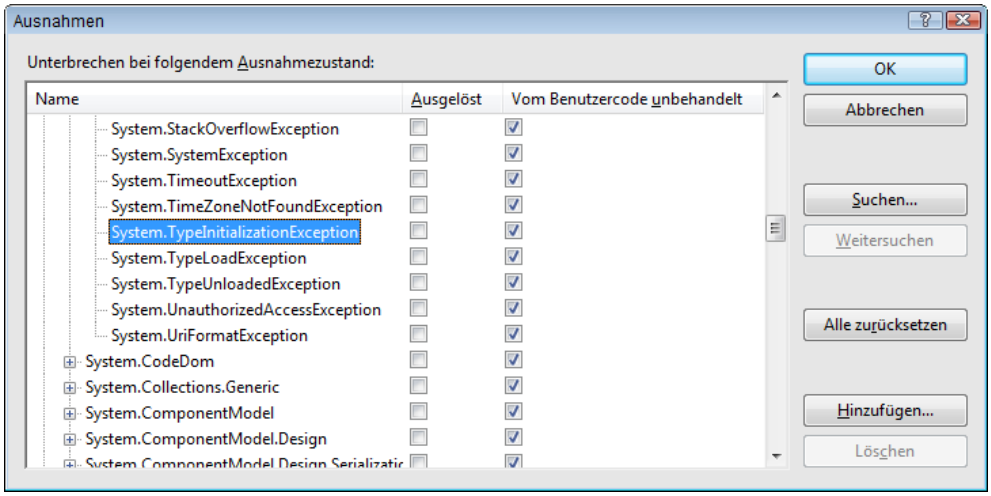
11.4.4 Das Standardverhalten bei Ausnahmen festlegen

Ganz kurz hatten wir ihn schon angesprochen, den Menüpunkt *Debuggen|Ausnahmen...*, hinter dem sich die in der folgenden Abbildung gezeigte Dialogbox versteckt.

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.

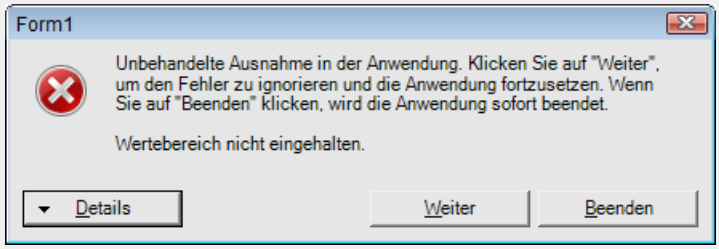
BEISPIEL 11.21: Eine Prozedur *Test* erwartet zwei Argumente *a* und *b*, die einen festgelegten Wertebereich haben. Wird dieser überschritten, löst die Prozedur eine Exception aus.

VB

```
Private Sub Test(a As Integer, b As Integer)
    If (a > 80000) Or (b > 100000) Then
      Throw New System.Exception("Wertebereich nicht eingehalten")
    End If
End Sub
```

Ergebnis

Wird die Prozedur mit ungültigen Parametern gestartet, erhalten Sie folgende Meldung:



11.4.7 Eigene Fehlerklassen

So schön auch die bisherigen Varianten sind, in einigen Fällen möchte man doch eigene Fehlerklassen erzeugen, sei es, dass spezielle Parameter übergeben werden oder 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.

VB

Die Klasse (wir leiten von der *Exception*-Klasse ab):

```
Public Class CMyError
Inherits Exception
```

Der Konstruktor:

```
    Public Sub New(message As String)
        MyBase.New()
    End Sub
```

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.

Eine zusätzliche Methode zum Versenden der E-Mail:

```
    Public Sub SendMailMessage()
        System.Diagnostics.Process.Start("mailto:support@nirgendwo.de?subject" &
                                                    "=Fehler&Body=" & Me.Message)
    End Sub
End Class
```

Selbstverständlich könnten Sie hier auch weitere Parameter definieren, zum Beispiel eine Zieladresse etc.

Die Verwendung (Auslösen der neuen Exception):

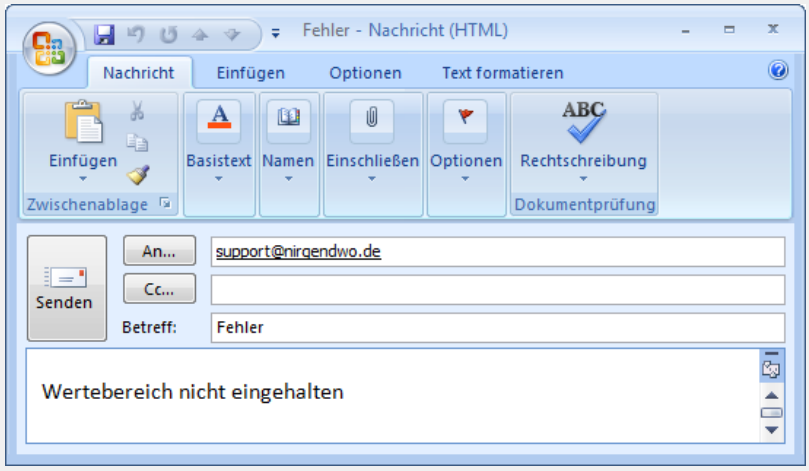
```
Private Sub test1(a As Integer, b As Integer)
   If (a > 80000) Or (b > 100000) Then _
                                  Throw New CMyError("Wertebereich nicht eingehalten")
End Sub
```

Die Verwendung (Fehlerbehandlung):

```
Private Sub Button8_Click(sender As Object, e As EventArgs) Handles Button8.Click
     Try
         Test1(77777777, 8)
     Catch mye As CMyError
         mye.SendMailMessage()
     End Try
 End Sub
```

Ergebnis

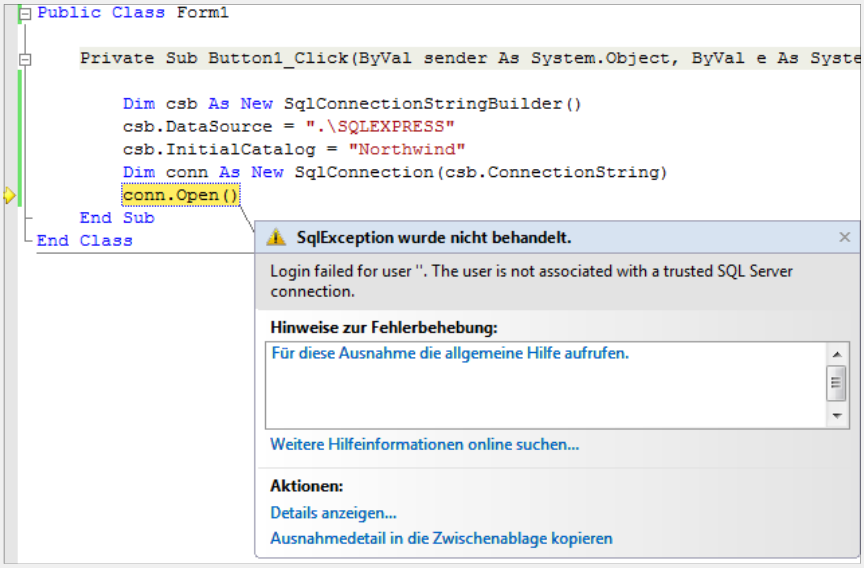
Nach dem Aufruf der Methode *SendMailMessage* öffnet sich der aktuelle E-Mail-Client, die Meldungstexte und die Adresse sind bereits eingetragen.



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).

BEISPIEL 11.23: Exception-Dialog beim Versuch, eine Verbindung zum SQL Server herzustellen



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

Erst mit .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 11.24: Eine Methode zur Division von zwei Gleitkommazahlen

VB

```
Public Function Dividieren(divident As Double, divisor As Double) As Double
```

Der Divisor darf nicht gleich 0 sein:

```
    System.Diagnostics.Contracts.Contract.Assume(divisor <> 0)
    Return divident / divisor
End Function
```

In älteren .NET-Versionen hätte man das allerdings auch mittels der *System.Diagnostics.Debug.-Assert()*-Methode realisieren können (siehe Seite 629). 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.

Kapitel 12

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 Kapitels.

12.1 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.

12.1.1 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.

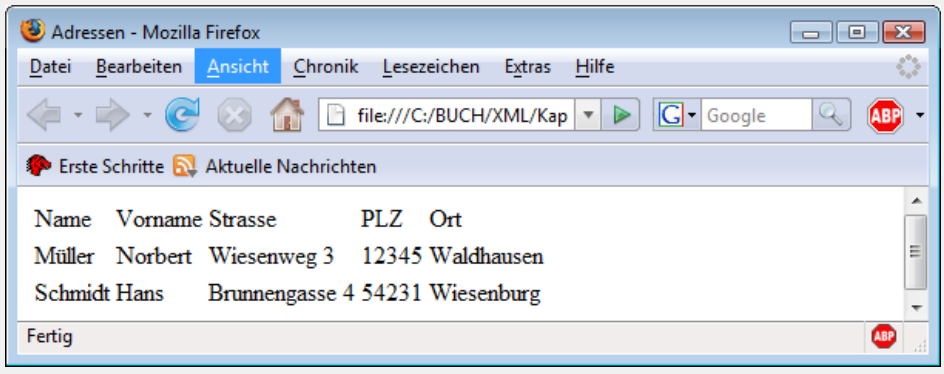
Das einfache Beispiel einer Adressliste soll den Unterschied verdeutlichen. In einer HTML-Datei würden auch Sie die Informationen sicherlich in Form einer Tabelle darstellen.

BEISPIEL 12.1: 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



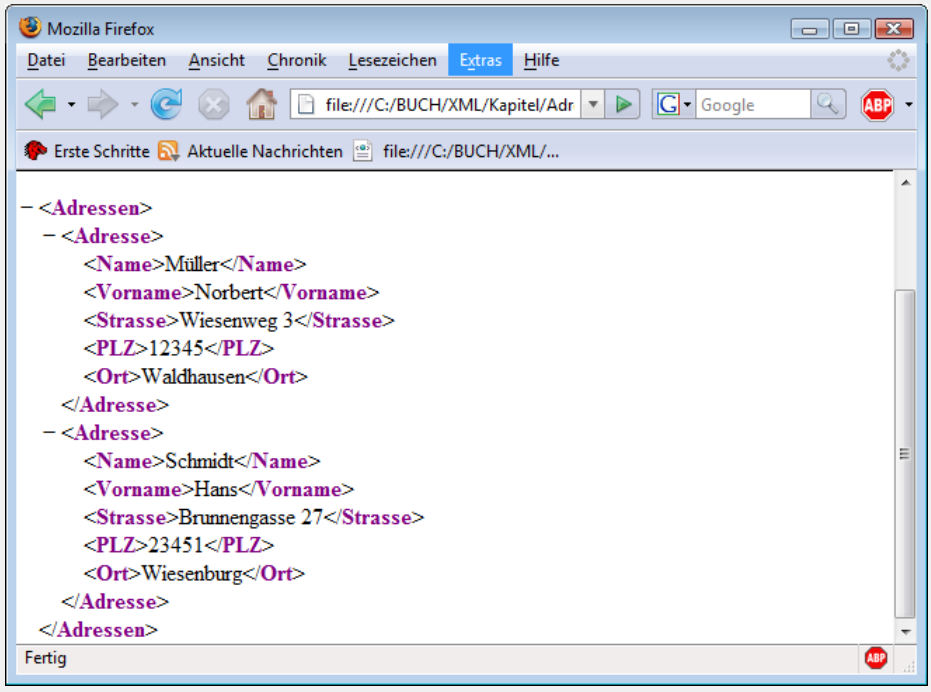
Die obigen Anweisungen sagen lediglich etwas über die Gestaltung der HTML-Seite aus. Einen Zusammenhang zwischen der Tabellenzelle "Name" und dem Inhalt "Müller" kann ein Programm nur schwer herstellen. Eine automatisierte Analyse und Weiterverarbeitung der Datei wird damit fast unmöglich.

BEISPIEL 12.2: 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 12.3: 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.

12.1.2 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 12.4: Prolog

XML
```
<?xml version="1.0" standalone="yes" ?>
```

BEISPIEL 12.5: 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 12.6: 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 12.7: Ein Element (*Vorname* ist der Elementname, *Norbert* ist der Elementinhalt)

```xml
<Vorname>Norbert</Vorname>
```

BEISPIEL 12.8: Ein Attribut (*Hauptwohnsitz* ist der Attributname, *Ja* ist der Wert)

```xml
   <Adresse Hauptwohnsitz="JA">
      <Name>Müller</Name>
      <Vorname>Norbert</Vorname>
   </Adresse>
```

BEISPIEL 12.9: 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.

12.1.3 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 12.10: 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 12.10: Ein wohlgeformtes XML-Dokument

```
   </ERDE>
</DATEN>
```

BEISPIEL 12.11: 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.

12.1.4 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 12.12: Einfache Angabe der Versionsnummer

XML
```
<?xml version="1.0"?>
```

BEISPIEL 12.13: 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 12.14: Keine externe DTD vorhanden

XML
```
<?xml version="1.0" standalone="yes" ?>
```

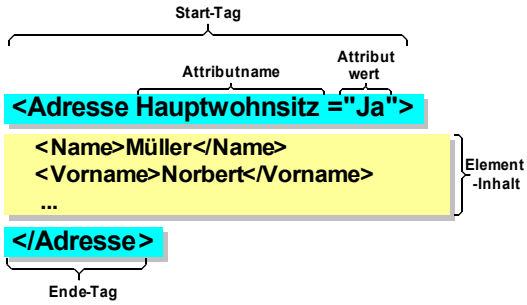
BEISPIEL 12.15: Eine externe DTD (welt.dtd) ist vorhanden

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.

12.1.5 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 12.16: 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 12.17: Richtige Deklaration

XML
```
<Anschrift>
  <Name>Müller</Name>
  <Vorname>Norbert</Vorname>
  <Vorname>Hans</Vorname>
  <Vorname>Werner</Vorname>
</Anschrift>
```

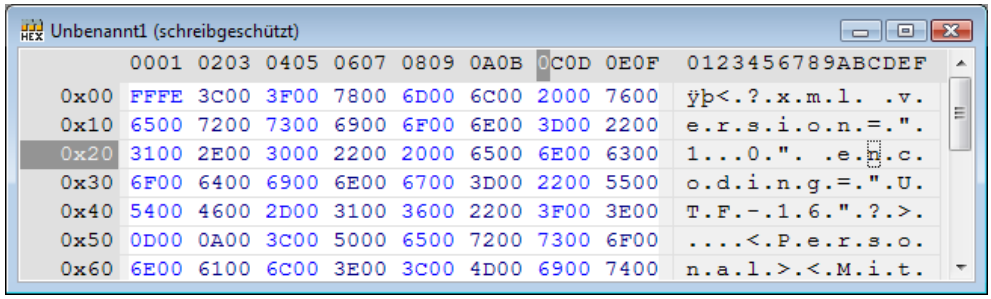
HINWEIS: Beachten Sie die Vorschrift, dass jedes XML-Dokument über genau ein Wurzel-Element (Root) verfügen muss.

12.1.6 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:

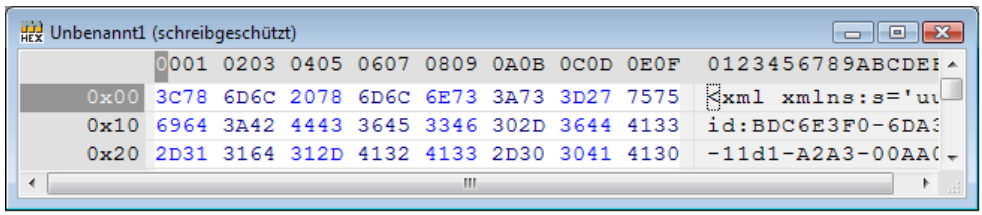
Eine UTF-16-Datei im Hex-Editor:



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

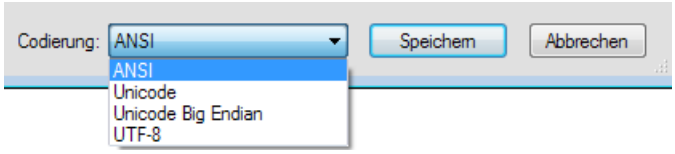
[1] Bei Unicode-Zeichen kann das höherwerte Byte das erste Byte (Little-Endian) oder das zweite Byte (Big-Endian) sein.

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.

12.2 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.

12.2.1 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 12.18: 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 12.18: 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:

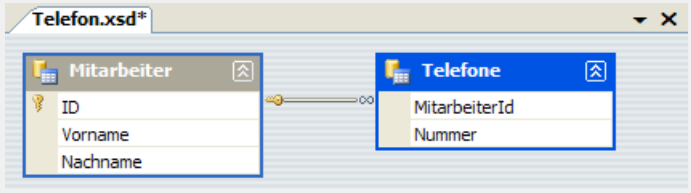
```
  <xs:key name="TelefonKey1" msdata:PrimaryKey="true">
    <xs:selector xpath=".//mstns:Mitarbeiter" />
    <xs:field xpath="@ID" />
  </xs:key>
```

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 12.19: 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.

12.2.2 XML-Schemas in Visual Studio analysieren

Vielleicht kann sich der eine oder andere noch daran erinnern, dass es in Visual Studio früher 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 2015 verfügt, wie auch die Vorgängerversion, über **keinen** grafischen Editor für XSD-Schemas, auch wenn die Bezeichnung einiger Menüpunkte mit "XML-Schema-Designer" etwas anderes suggeriert!

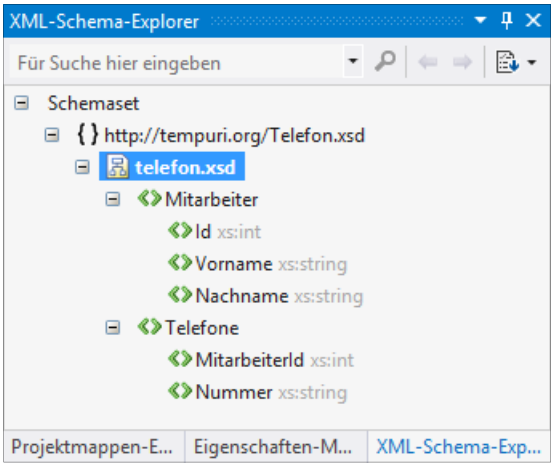
Doch was bietet Visual Studio 2015 dem Entwickler in Bezug auf XML-Schemas? Im Wesentlichen handelt es sich um verschiedene Varianten von Betrachtern, mit denen Sie vorhandene Schemadefinitionen grafisch ansprechend darstellen können:

- 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.

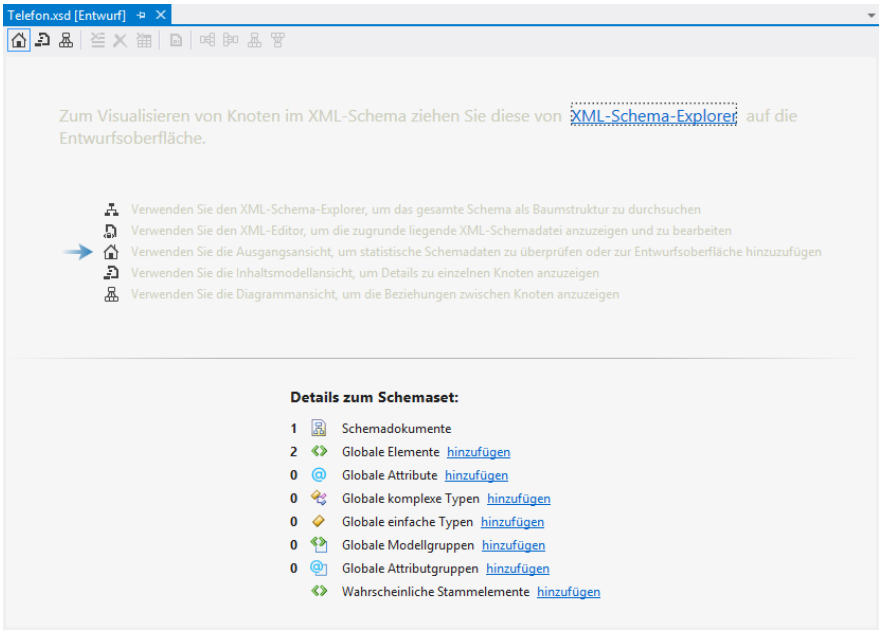
In einer hierarchischen Strukturansicht werden neben dem aktuellen Schema auch alle zugehörigen XSD-Dateien, d h. eingeschlossene oder importierte Schemas, angezeigt. Zu besseren Übersicht lassen sich Details ein-/ausblenden, Sortierfolgen ändern oder direkt Einträge suchen.

HINWEIS: Klicken Sie doppelt auf einen der Einträge im XML-Schema-Explorer 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-Designer

Öffnen Sie eine XML-Schema-Datei 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.

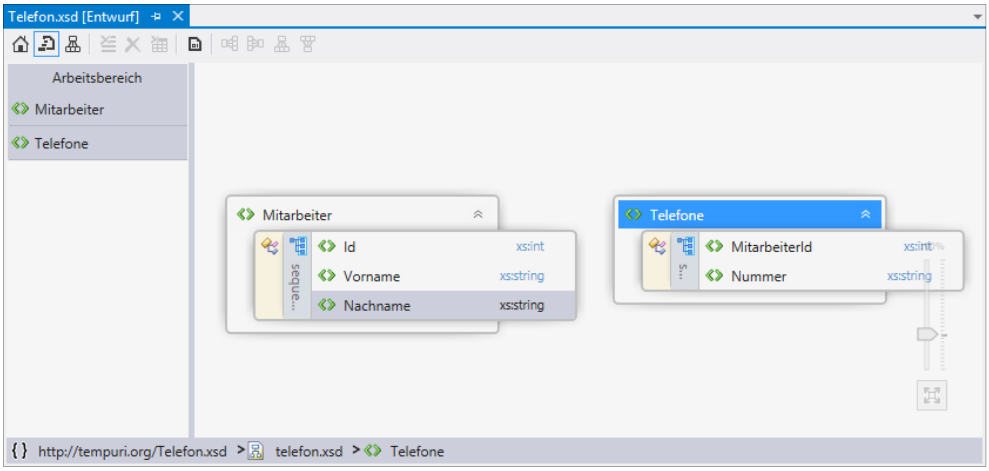


HINWEIS: An dieser Stelle sieht man auch wieder mal wie sinnvoll das "Trauerlayout" von Visual Studio ist, die Schrift (Grau auf Grau) ist kaum zu erkennen.

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:



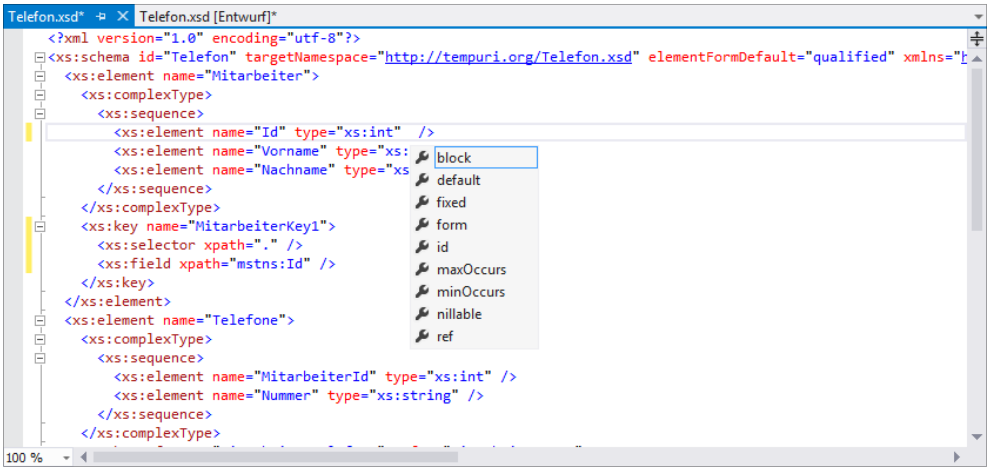
HINWEIS: Weitere Detaildaten, wie zum Beispiel Min/Max-Werte oder Default-Werte können Sie im Eigenschaftenfenster einsehen.

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

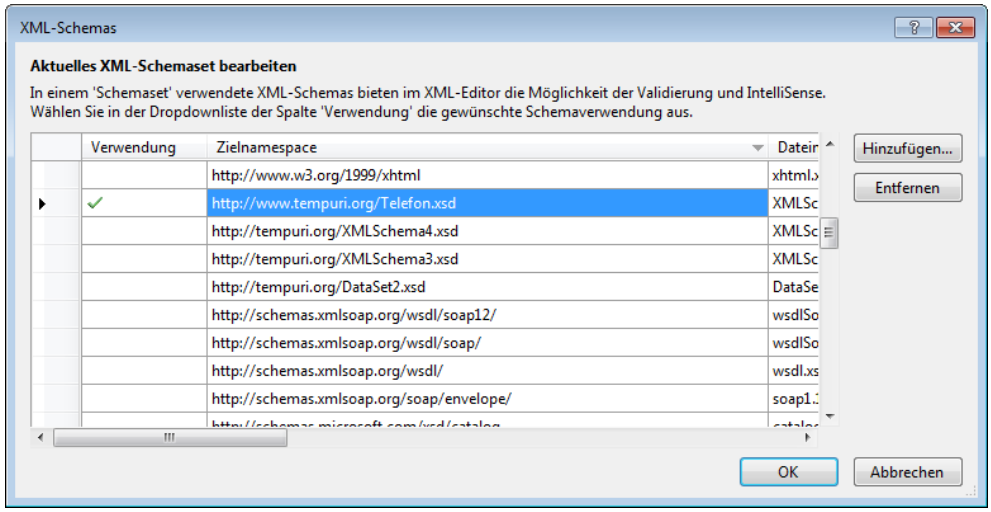
Last but not least, gibt es natürlich auch eine Möglichkeit, Schema-Dateien zu editieren, doch hier haben wir es mit einem recht einfachen XML-Texteditor zu tun, umfangreiche Schemadefinitionen werden Sie mit diesem Editor wohl kaum erstellen wollen, die Übersicht leidet sicher schon nach wenigen Zeilen, und der XML-Schema-Explorer gibt auch keine große Hilfestellung, ganz zu schweigen davon, dass Sie auch detaillierte XSD-Kenntnisse benötigen.



12.2.3 XML-Datei mit XSD-Schema erzeugen

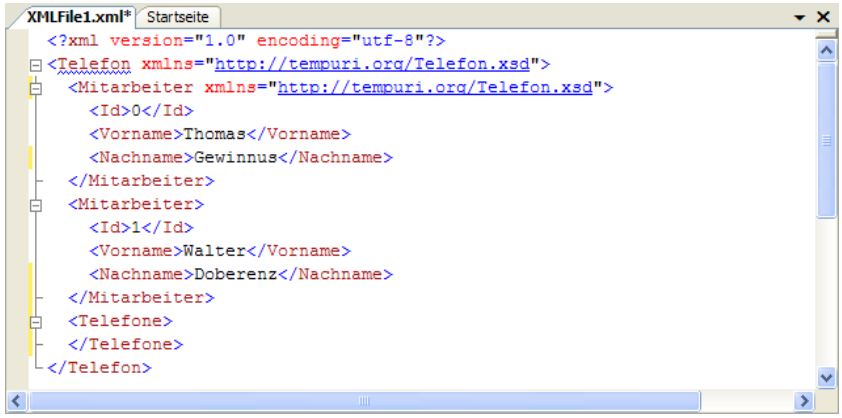
Erstellen Sie eine neue XML-Datei (Menüpunkt *Projekt/Neues Element hinzufügen*). Nach dem Öffnen dieser Datei können Sie die Eigenschaften des Dokuments bearbeiten. Für uns ist die Eigenschaft *Schemata* interessant. Weisen Sie hier das bereits erstellte XSD-Schema zu[1]:

[1] Dazu müssen Sie bei den gewünschten Schemas 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:



12.2.4 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.

12.3 XML-Integration in Visual Basic

Mit LINQ to XML (vormals XLINQ) ist eine komfortable Unterstützung für die Verarbeitung von XML-Dokumenten direkt in Visual Basic integriert.

XML-Dokumente sind jetzt nicht mehr "Fremdkörper", auf die mit extra APIs und externen Objektmodellen zugegriffen werden muss, sondern der 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, die bisher mit mit dem XML DOM (Namespace *System.Xml*) etc. gearbeitet haben, besteht kein Grund zur Sorge: Ihr Quellcode funktioniert nach wie vor, Sie können sukzessiv den Quellcode migrieren oder mit beiden Technologien arbeiten.

Daneben hielten bereits mit Visual Basic 9 auch diverse Neuerungen bei der XML-Unterstützung Einzug, die Sie in dieser Form nicht bei C# finden werden[2]. So bietet Ihnen Visual Basic zusätzlich die folgenden Möglichkeiten:

- Einbetten von XML-Code in den Visual Basic-Code mittels XML-Literal
- Navigieren in XML-Dokumenten mit den XML-Achsen-Eigenschaften

auf die wir zunächst eingehen werden, bevor wir uns der LINQ to XML-API zuwenden.

12.3.1 XML-Literale

Wer schon XML-Dokumente in früheren Versionen von Visual Basic erzeugt hat, wird sicher die leidvolle Erfahrung gemacht haben, dass es umständlicher kaum geht. Im besten Fall verwenden Sie einen *XmlWriter,* mit dessen Hilfe Sie die einzelnen XML-Knoten mühevoll, syntaktisch richtig und in der korrekten Reihenfolge erzeugen müssen. Mit dem Schreiben von XML-Code, wie er später ausgegeben wird, hat dies wenig zu tun.

Demgegenüber bietet Visual Basic jetzt die Möglichkeit, XML-Code, so wie Sie ihn kennen, direkt in den Basic-Code zu integrieren.

BEISPIEL 12.20: Was halten Sie zum Beispiel von folgendem Code?

VB

```
Dim Stammdaten As XElement = <Stammdaten>
                                 <Kunden>
                                     <Kunde id="1">
```

[1] Wobei derartige Kenntnisse niemandem schaden.

[2] Sie wussten ja schon immer, dass VB besser ist ...

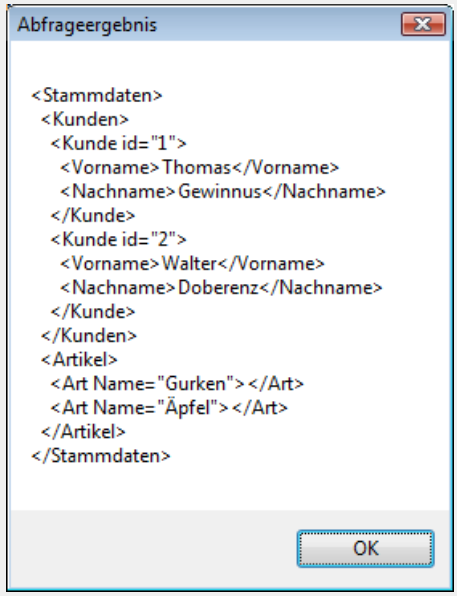
BEISPIEL 12.20: Was halten Sie zum Beispiel von folgendem Code?

```
                                        <Vorname>Thomas</Vorname>
                                        <Nachname>Gewinnus</Nachname>
                                    </Kunde>
                                    <Kunde id="2">
                                        <Vorname>Walter</Vorname>
                                        <Nachname>Doberenz</Nachname>
                                    </Kunde>
                                </Kunden>
                                <Artikel>
                                    <Art Name="Gurken"></Art>
                                    <Art Name="Äpfel"></Art>
                                </Artikel>
                            </Stammdaten>
MessageBox.Show(Stammdaten.ToString(), "Abfrageergebnis")
```

Sicher ist Ihnen nicht entgangen, dass auf den Unterstrich (_) als Zeilenendzeichen verzichtet wurde. Das ist kein Druckfehler, innerhalb des XML-Codes können Sie darauf verzichten.

Ergebnis

Die Anzeige in der MessageBox zeigt ein "reinrassiges" XML-Dokument:



Das sieht wie XML-Code aus, ist leicht lesbar, verständlich und, last but not least, es werden auch noch diverse Fehlerprüfungen durch den Compiler durchgeführt, um ein korrektes XML-Fragment zu erhalten.

Doch was macht eigentlich der Compiler aus obigem XML-Code? Mit Hilfe des .NET-Reflectors wagen wir einen Blick in die Assembly und siehe da, hier findet sich ganz normaler Code mit entsprechenden LINQ to XML-Objekten (wir gehen ab Abschnitt 12.3.3 näher darauf ein):

```
...
    Dim VB$t_ref$S0 As New XElement(XName.Get("Stammdaten", ""))
    Dim VB$t_ref$S1 As New XElement(XName.Get("Kunden", ""))
    Dim VB$t_ref$S2 As New XElement(XName.Get("Kunde", ""))
    Dim VB$t_ref$S3 As New XAttribute(XName.Get("id", ""), "1")
    VB$t_ref$S2.Add(VB$t_ref$S3)
    Dim VB$t_ref$S4 As New XElement(XName.Get("Vorname", ""))
    VB$t_ref$S4.Add("Thomas")
    VB$t_ref$S2.Add(VB$t_ref$S4)
    VB$t_ref$S4 = New XElement(XName.Get("Nachname", ""))
    VB$t_ref$S4.Add("Gewinnus")
    VB$t_ref$S2.Add(VB$t_ref$S4)
    VB$t_ref$S1.Add(VB$t_ref$S2)
...
    MessageBox.Show(VB$t_ref$S0.ToString, "Abfrageergebnis")
...
```

Wem das alles noch nicht genug ist, für den haben wir noch ein weiteres Feature auf Lager: Sie können in diesen XML-Code Platzhalter einfügen, die wiederum VB-Code aufnehmen können.

BEISPIEL 12.21: Einfügen eines Platzhalters in den XML-Code

VB

```
Dim art1 As String = "Gurken"
Dim Stammdaten2 As XElement = <Artikel>
                                  <Art Name=<%= art1 %>></Art>
                                  <Art Name="Äpfel"></Art>
                              </Artikel>
MessageBox.Show(Stammdaten2.ToString(), "Abfrageergebnis")
```

Ergebnis

Alternativ könnten Sie in den Platzhalter auch eine Funktion einfügen, die zum Beispiel einen kompletten Knoten zurückgibt.

BEISPIEL 12.22: Funktionsergebnis im Platzhalter

VB

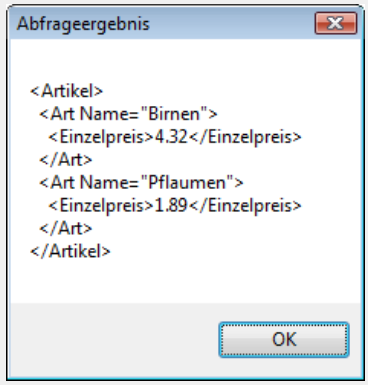
```
    Function NeuerArtikel(Name As String, Preis As Single) As XElement
        NeuerArtikel = <Art Name=<%= Name %>><Einzelpreis><%= Preis %></Einzelpreis></Art>
    End Function
...
```

BEISPIEL 12.22: Funktionsergebnis im Platzhalter

```
Dim Stammdaten2 As XElement = <Artikel>
                                <%= NeuerArtikel("Birnen", 4.32) %>
                                <%= NeuerArtikel("Pflaumen", 1.89) %>
                              </Artikel>
MessageBox.Show(Stammdaten2.ToString(), "Abfrageergebnis")
```

Ergebnis

Der erzeugte XML-Baum:



12.3.2 Einfaches Navigieren durch späte Bindung

"Wenn schon XML im Quellcode, dann richtig!" Dieses Motto dürfte auf die folgenden Möglichkeiten uneingeschränkt zutreffen.

Neben dem einfachen Erzeugen von XML-Code wird auch die Navigation innerhalb dieser Dokumente wesentlich vereinfacht. Über sogenannte XML-Achsen-Eigenschaften (neudeutsch *Xml axis properties*) die durch späte Bindung der XML-Bezeichner an korrespondierende XML-Attribute bzw. Elemente zur Laufzeit erfolgen, können Sie auf die untergeordneten-Elemente sowie die Attribute zugreifen.

Folgende Möglichkeiten bestehen:

- Zugriff auf ein untergeordnetes Elemente per *.<Name des Elements>*
- Zugriff auf ein Attribut per *.@Name des Attributs*
- Zugriff auf ein untergeordnetes Element (beliebige Tiefe) mit vorgegebenem Namen mit *...<Name des Elements>*
- Zugriff auf die Werte des ersten Knotens per *Value*-Eigenschaft
- Verwendung der Auflistung, um per Index auf ein Element zuzugreifen.

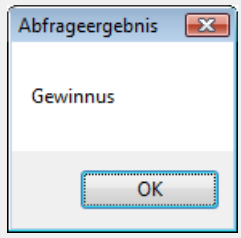
HINWEIS: All diese netten Features werden vom Compiler intern in entsprechende Methoden-Aufrufe konvertiert, der Visual Basic-Editor liefert also hier "nur" etwas fürs Auge und die Bequemlichkeit (*Eye candy*).

Einige kurze Beispiele sorgen für mehr Klarheit. Ausgangspunkt ist das auf Seite 668 gezeigte XML-Dokument.

BEISPIEL 12.23: Zugriff auf untergeordnete Elemente

VB
```
MessageBox.Show(Stammdaten.<Kunden>.<Kunde>.<Nachname>.Value, "Abfrageergebnis")
```

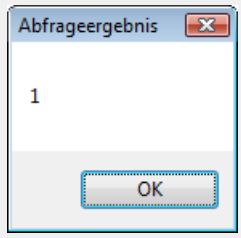
Ergebnis



BEISPIEL 12.24: Zugriff auf Attribute

VB
```
MessageBox.Show(Stammdaten.<Kunden>.<Kunde>.@id, "Abfrageergebnis")
```

Ergebnis



BEISPIEL 12.25: Zugriff auf untergeordnete Elemente

VB
```
For Each s In Stammdaten...<Nachname>
   ListBox1.Items.Add(s.Value)
Next
```

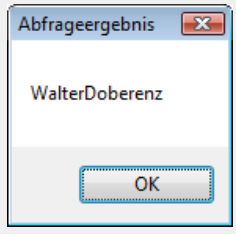
Das Beispiel zeigt **beide** Nachnamen im Listenfeld an.

BEISPIEL 12.26: Zugriff auf ein spezifisches Element per Index

VB
```
MessageBox.Show(Stammdaten.<Kunden>.<Kunde>(1).Value, "Abfrageergebnis")
```

Ergebnis

... zeigt das zweite! Element <Kunde> an (die Liste beginnt bei 0):

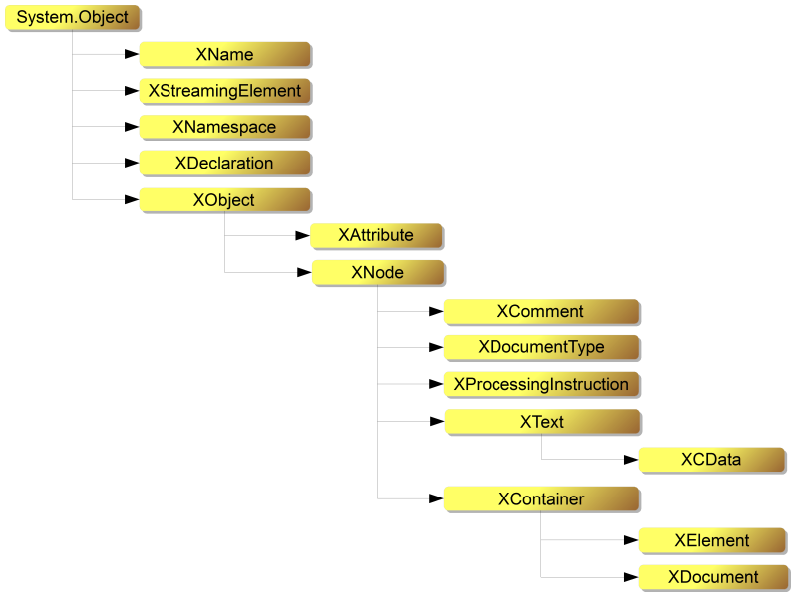


HINWEIS: Doch Achtung: Verwenden Sie einen Index, der die Länge der Liste überschreitet, ist "Schluss mit lustig", Sie werden mit einer *NullReference*-Exception traktiert.

Da Sie es mit einer Liste zu tun haben, können Sie selbstverständlich auch Gebrauch von deren Eigenschaften machen. So lässt sich beispielsweise mit *Count* die Anzahl der Datensätze (Knoten) abrufen.

12.3.3 Die LINQ to XML-API

Vor einigen Jahren wurde auch eine neue API für den Zugriff auf XML-Dokumente eingeführt, die unter anderem 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.



In der folgenden Tabelle finden Sie eine kurze Beschreibung zu den wichtigsten neuen XML-API-Klassen.

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 eigenlichen 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

Klasse	Beschreibung
	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 manipulieren oder zu entfernen. Über den Konstruktor können Sie komplette XML-Baumstrukturen erzeugen, ohne auch nur eine Methode aufzurufen 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.

HINWEIS: Bei den Klassen *XNode*, *XContainer* und *XObject* handelt es sich um abstrakte Klassen, Sie können diese also nicht direkt verwenden.

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.

12.3.4 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 12.27: Sie möchten folgendes XML-Dokument erzeugen:

VB

```
<Artikeliste>
 <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>
   <Einzelpreis>19</Einzelpreis>
   <Lagerbestand>17</Lagerbestand>
 </Artikel>
</Artikelliste>
```

Die Umsetzung des Beispiels in Visual Basic macht von den Möglichkeiten der XML-Literale intensiv Gebrauch und dürfte die optimale Lösung darstellen, handelt es sich doch um die direkte Zuweisung von XML-Code an das Objekt:

```
Imports System.Xml.Linq

Public Class Form1
    Private ArtikelListe As XElement
    ...
        ArtikelListe = <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>
                               <Einzelpreis>19</Einzelpreis>
                               <Lagerbestand>17</Lagerbestand>
                           </Artikel>
                       </Artikelliste>
...
```

Selbstverständlich hilft Ihnen auch hier die Syntax-Prüfung beim Erstellen fehlerfreier Dokumente.

HINWEIS: Für variable Inhalte im XML-Code können Sie auch Platzhalter verwenden, wir kommen an späterer Stelle darauf zurück.

Alternativ steht Ihnen in Visual Basic auch die Variante mit den geschachtelten Konstruktoren zur Verfügung:

```
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 ähnlich 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 12.28: Deklaration und Kommentar in ein *XDocument*-Objekt einfügen

VB
```
Dim mydoc As New XDocument(
                    New XDeclaration("1.0", "utf-8", "yes"),
                    New XComment("Artikeldatenbank"),
                    New XElement("Artikelliste")
                    ...
```

12.3.5 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 nahe liegende 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:

```
Load(String)
Load(TextReader)
Load(XmlReader)
Load(String, LoadOptions)
Load(TextReader, LoadOptions)
```

```
Load(XmlReader, LoadOptions)

Parse(String)
Parse(String, LoadOptions)
```

Alternativ bietet sich die Methode *Parse()* an, um die Daten aus einer Zeichenkette einzulesen.

BEISPIEL 12.29: XML-Inhalte aus einer Datei laden

VB
```
Dim ArtikelListe As XElement = XElement.Load("Artikel.xml")
```

BEISPIEL 12.30: Laden aus einer Zeichenkette

VB
```
Dim s As String = "<Artikelliste><Artikel><Name>Junger Hund</Name>" &
                  "<Preis>120.17</Preis></Artikel></Artikelliste>"

Dim ArtikelListe As XElement = XElement.Parse(s)
MessageBox.Show(ArtikelListe.ToString(), "Artikelliste")
```

Ergebnis

Die angezeigten Daten:

Das Laden aus einer konstanten Zeichenkette gestaltet sich in VB dank XML-Literalen etwas einfacher:

```
Dim ArtikelListe As XElement = <Artikelliste><Artikel><Name>Junger Hund</Name>
                               <Preis>120.17</Preis></Artikel></Artikelliste>
```

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:

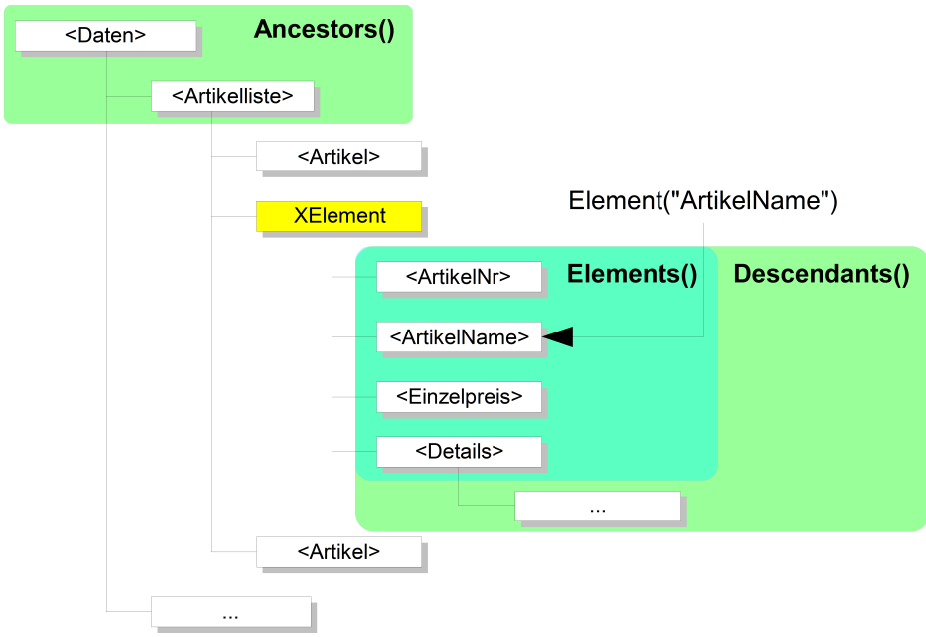
```
Save(String)
Save(TextWriter)
Save(XmlWriter)
Save(String, SaveOptions)
Save(TextWriter, SaveOptions)
```

BEISPIEL 12.31: Sichern in einer XML-Datei

```
VB ArtikelListe.Save("ArtikelNeu.xml")
```

12.3.6 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 Tabelle zeigt weitere Methoden und deren Beschreibung:

Methode	Beschreibung
Ancestors	Rekursive *XElement*-Liste aller übergeordneten Elemente des aktuellen Knotens.
CompareDocumentOrder	Der relative Abstand zweier Knoten (*XNode*).
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.

Methode	Beschreibung
IsAfter	Liefert *True*, wenn sich der aktuelle Knoten hinter dem Vergleichsknoten befindet.
IsBefore	Liefert *True*, wenn sich der aktuelle Knoten vor dem Vergleichsknoten befindet.
Nodes	Liefert eine Liste der untergeordneten Knoten.

BEISPIEL 12.32: Anzeige aller Artikelnamen in einem Listenfeld

VB

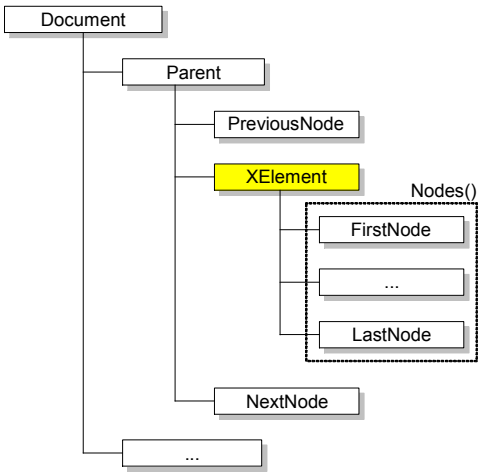
Wir machen von der späten Bindung Gebrauch, der Quellcode entspricht damit weitgehend der XML-Darstellung:

```
Dim ArtikelListe As XElement
ArtikelListe = XElement.Load("Artikel.xml")
For Each Artikel In ArtikelListe.<Artikel>
    ListBox1.Items.Add("" & Artikel.<Artikelname>.Value())
Next
```

Haben Sie Ihrem Visual Basic-Projekt auch die zugehörige Schemadatei zugeordnet, können Sie noch etwas bequemer arbeiten, der Editor bietet Ihnen bereits die möglichen Elemente per Intellisense an:

```
' Listbox füllen
For Each Artikel In ArtikelListe.<Artikel>
    ListBox1.Items.Add("" & Art                    me>.Value())
Next
' Anzahl bestimmter Elemente be
MessageBox.Show(ArtikelListe.<A                    .ToString & " Artikel in der Liste")
```


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, im Zusammenhang mit dem XML-DOM kommen wir jedoch darauf zurück.

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 12.33: Neue Liste mit 20 Artikeln bilden, ohne die ersten 5 Einträge

VB
```
Dim Artikelliste2 = ArtikelListe.<Artikel>.Skip(5).Take(20)
```

12.3.7 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 12.34: Filtern der Artikelliste (Einzelpreis < 20)

VB
```
Dim ArtikelListe As XElement
ArtikelListe = XElement.Load("Artikel.xml")
Dim Artikelliste3 = ArtikelListe.<Artikel>.Where(
                                Function(a) CSng(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 12.35: Filtern der Artikelliste (Einzelpreis < 20)

VB
```
Dim Artikelliste3 = From a In ArtikelListe.<Artikel>
                    Where CSng(a.Element("Einzelpreis")) < 20
```

Auf die umfangreichen Möglichkeiten, Filterbedingungen zusammenzustellen, wollen wir an dieser Stelle nicht weiter eingehen, dazu wurden bereits im Kapitel 6 die Grundlagen gelegt.

12.3.8 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 12.36: Editieren eines Elements

VB

```
Dim Artikel = ArtikelListe.<Artikel>(1)
MessageBox.Show("Aktueller Artikel: " & Artikel.<Artikelname>.Value)
MessageBox.Show(Artikel.ToString(), "XML")
Artikel.SetElementValue("Einzelpreis", 200)
MessageBox.Show(Artikel.ToString(), "XML")
```

Hinzufügen von Daten

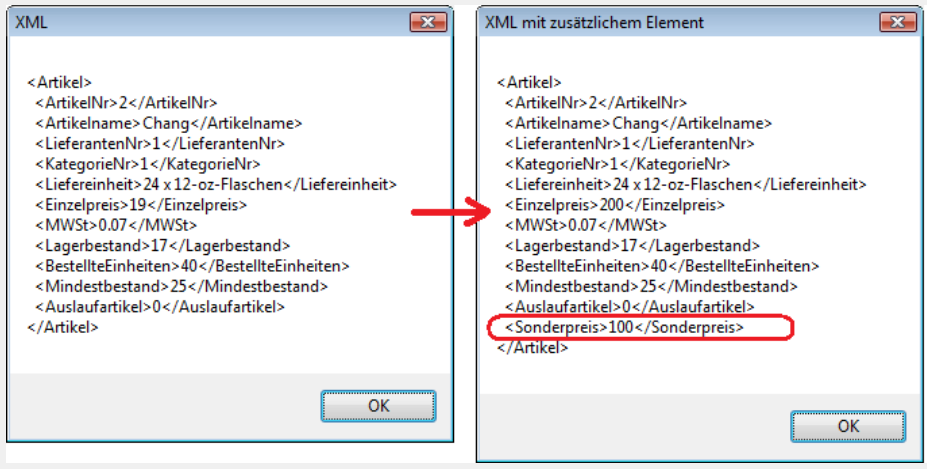
Mit den Methoden *Add*, *AddFirst*, *AddBeforeSelf*, *AddAfterSelf* können Sie neue Elemente hinzufügen.

BEISPIEL 12.37: Hinzufügen eines untergeordneten Elements

VB

```
Dim Artikel = ArtikelListe.<Artikel>(1)
Artikel.Add(New XElement("Sonderpreis", "100"))
```

Ergebnis



BEISPIEL 12.38: Hinzufügen eines kompletten XML-Fragments (Artikel) in die Artikelliste

VB

```
ArtikelListe.Add(<Artikel>
                    <ArtikelNr>2317</ArtikelNr>
                    <Artikelname>Junger Hund</Artikelname>
                    <LieferantenNr>1</LieferantenNr>
                    <KategorieNr>1</KategorieNr>
                    <Liefereinheit>10 Kartons x 20 Dosen</Liefereinheit>
                    <Einzelpreis>18</Einzelpreis>
```

BEISPIEL 12.38: Hinzufügen eines kompletten XML-Fragments (Artikel) in die Artikelliste

```
                    <MWSt>0.07</MWSt>
                    <Lagerbestand>18</Lagerbestand>
                    <BestellteEinheiten>0</BestellteEinheiten>
                    <Mindestbestand>10</Mindestbestand>
                    <Auslaufartikel>0</Auslaufartikel>
                </Artikel>)
```

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.

12.3.9 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 VB-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 Feldes und Projektion der XML-Elemente in -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 12.39: XML-Dokumente transformieren

VB

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.

```
Dim ArtikelListe As XElement

ArtikelListe = XElement.Load("Artikel.xml")
Dim Abfrage As XElement = <Artikel>
                              <%= From artikel In ArtikelListe.<Artikel>
                                  Where artikel.<ArtikelNr>.Value > 5
                                  Where artikel.<ArtikelNr>.Value < 20
                                  Select New XElement("Art",
                                     New XAttribute("Nr", artikel.<ArtikelNr>.Value),
                                     New XAttribute("Name", artikel.<Artikelname>.Value),
                                     New XAttribute("Wert", artikel.<Lagerbestand>.Value *
                                                    artikel.<Einzelpreis>.Value))
                               %>
                             </Artikel>                                  )
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 12.40: Wie in obigem Beispiel sollen die Daten abgefragt, jetzt jedoch im HTML-Format gesichert werden.

VB

```
...
   Dim html As XElement = <html><body><table>
                             <%= From artikel In ArtikelListe.<Artikel>
                                 Where artikel.<ArtikelNr>.Value > 5
                                 Where artikel.<ArtikelNr>.Value < 20
                                 Select New XElement("tr",
                                   New XElement("td", artikel.<ArtikelNr>.Value),
                                   New XElement("td", artikel.<Artikelname>.Value),
```

BEISPIEL 12.40: Wie in obigem Beispiel sollen die Daten abgefragt, jetzt jedoch im HTML-Format gesichert werden.

```
                                New XElement("td", artikel.<Lagerbestand>.Value *
                                 artikel.<Einzelpreis>.Value)
                               ) %>
                             </table></body></html>
    html.Save("artikel.html")
```

Ergebnis

Die ausgegebene Datei:

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
11	Queso Cabrales	462

BEISPIEL 12.41: Konvertieren in eine typisierte Objektliste zwecks Anzeige in einer *DataGridView*

VB

```
...
        Dim Abfrage = From artikel In ArtikelListe.Elements()
                      Where artikel.Element("ArtikelNr").Value > 5
                      Where artikel.Element("ArtikelNr").Value < 20
                      Select New With {.Nr = artikel.Element("ArtikelNr").Value,
                           .Name = artikel.Element("Artikelname").Value,
                           .Wert = (artikel.Element("Lagerbestand").Value *
                                    artikel.Element("Einzelpreis").Value)}
        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.

12.4 Verwendung des DOM unter .NET

Obwohl wir im vorhergehenden Abschnitt einfachere Wege zum Erzeugen/Verarbeiten von XML-Dateien vorgestellt haben, wollen wir Sie auch mit der "Low-Level"-Programmierung per DOM vertraut machen. So bietet sich die Möglichkeit, DOM-Codebeispiele aus anderen Sprachen schnell zu übernehmen.

12.4.1 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 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

Objekt	Beschreibung
XmlParseError	Objekt zur gezielten Fehleranalyse, das neben der Fehlerbeschreibung auch die Position und die Fehlernummer bereitstellt
XmlAttribute	Objekt zum Zugriff auf Element-Attribute
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

12.4.2 DOM-Integration in Visual Basic

Selbstverständlich ist auch in VB 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.

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.

12.4.3 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 12.42: Laden aus einer Datei

VB
```
Imports System.Xml
...
    Dim xmldoc As New XmlDocument()
    Try
      xmldoc.LoadXml("C:\test.xml")
```

BEISPIEL 12.42: Laden aus einer Datei

```
    MessageBox.Show("OK")
  Catch
    MessageBox.Show("Fehler")
  End Try
```

Alternativ können Sie die Dokumente auch aus einer *String*-Variablen laden. Verwenden Sie dazu die *LoadXML*-Methode.

BEISPIEL 12.43: Laden von XML-Daten aus einem String

VB

```
Dim xmldoc As New XmlDocument()
Dim mystr As String

mystr = "<WELT>"
mystr = mystr & "  <KONTINENTE>"
mystr = mystr & "  </KONTINENTE>"
mystr = mystr & "</WELT>"
Try
  xmldoc.LoadXml(mystr)
  MessageBox.Show("OK")
Catch
  MessageBox.Show("Fehler")
End Try
```

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.

12.4.4 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 12.44: Ein erstes XML-Dokument (Versuch 1)

VB

```
Dim xmldoc As New XmlDocument()
Dim root As XmlNode
```

Wir erzeugen ein Wurzel-Element ...

```
root = xmldoc.CreateElement("WELT")
```

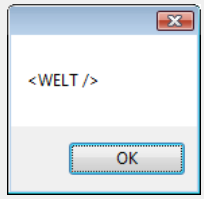
... und fügen dieses dem DOM-Dokument hinzu:

```
xmldoc.AppendChild(root)
MessageBox.Show(xmldoc.InnerXml)
```

BEISPIEL 12.44: Ein erstes XML-Dokument (Versuch 1)

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 12.45: Unser zweiter Versuch (Sichern in Datei)

VB

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
```

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:

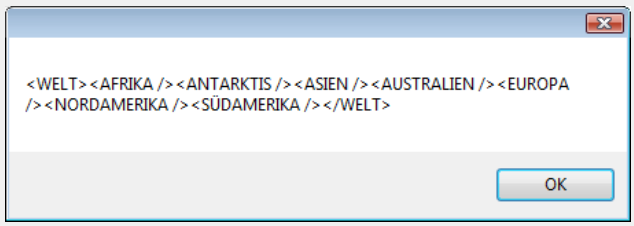
```
node = xmldoc.CreateElement("AFRIKA")
root.AppendChild(node)
node = xmldoc.CreateElement("ANTARKTIS")
root.AppendChild(node)
node = xmldoc.CreateElement("ASIEN")
root.AppendChild(node)
node = xmldoc.CreateElement("AUSTRALIEN")
root.AppendChild(node)
node = xmldoc.CreateElement("EUROPA")
root.AppendChild(node)
node = xmldoc.CreateElement("NORDAMERIKA")
root.AppendChild(node)
node = xmldoc.CreateElement("SÜDAMERIKA")
root.AppendChild(node)
```

Wir sichern die XML-Daten in einer Datei:

```
xmldoc.Save("c:\Test2.xml")
MessageBox.Show(xmldoc.InnerXml)
```

BEISPIEL 12.45: Unser zweiter Versuch (Sichern in Datei)

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.

12.4.5 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 12.46: Einlesen der eben erzeugten Datei und Anzeige des ersten Elements.

VB

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
xmldoc.Load("c:\test2.xml")
```

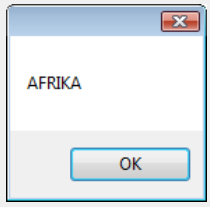
Einen "Zeiger" auf das Wurzelelement ("WELT") bestimmen:

```
root = xmldoc.DocumentElement
```

Nachfolgend können wir den ersten Untereintrag ermitteln und anzeigen:

```
node = root.FirstChild()
MessageBox.Show(node.Name)
```

Ergebnis



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-To*-Schleife.

Das Ergebnis ist in allen Fällen dasselbe.

BEISPIEL 12.47: Anzeige der einzelnen Elemente mit Hilfe der DOM-Eigenschaften und Methoden in einer *ListBox*.

VB

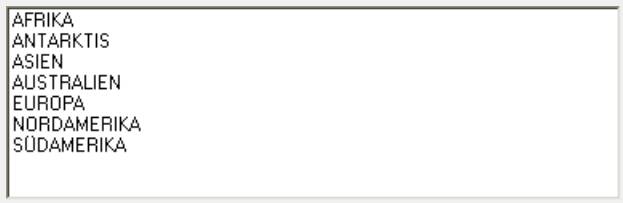
```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode

xmldoc.Load("c:\test2.xml")
ListBox1.Items.Clear()
root = xmldoc.DocumentElement
```

Auslesen mit Hilfe der Collection *ChildNodes:*

```
For Each node In root.ChildNodes
  ListBox1.Items.Add(node.Name)
Next
```

Ergebnis

Die Ausgabe in der *ListBox:*



12.4.6 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 12.48: Verwendung von *SelectSingleNode*

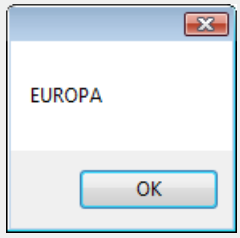
VB

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
xmldoc.Load("c:\test2.xml")
root = xmldoc.DocumentElement
node = root.SelectSingleNode("EUROPA")
MessageBox.Show(node.Name)
```

BEISPIEL 12.48: Verwendung von *SelectSingleNode*

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.

12.4.7 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 12.49: Attribute und Elemente verwenden

VB

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.

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
xmldoc.Load("c:\test2.xml")
```

Auswahl eines bestimmten Knotens:

```
root = xmldoc.DocumentElement
node = root.SelectSingleNode("EUROPA")
```

Einfügen von Zusatzinformationen:

```
With node.AppendChild(xmldoc.CreateElement("Fläche"))
  .InnerText = "10500000"
End With
With node.AppendChild(xmldoc.CreateElement("Einwohner"))
  .InnerText = "718500000"
End With
```

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.

BEISPIEL 12.49: Attribute und Elemente verwenden

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("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 12.50: Erweitern der Einträge von *Deutschland* und *Frankreich* um die Einträge *Fläche* und *Einwohner*

VB

```
    Dim xmldoc As New XmlDocument()
    Dim root, node As XmlNode
...
```

BEISPIEL 12.50: Erweitern der Einträge von *Deutschland* und *Frankreich* um die Einträge *Fläche* und *Einwohner*

```
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:

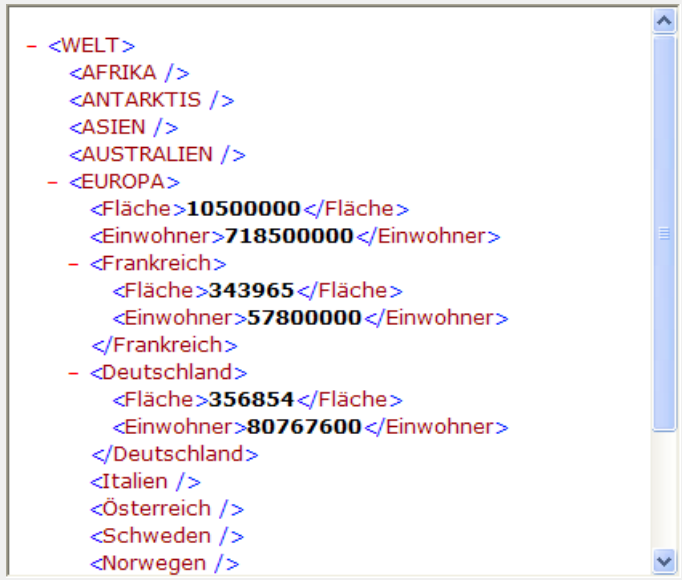
```
node.AppendChild(xmldoc.CreateElement("Fläche")).InnerText = "356854"
node.AppendChild(xmldoc.CreateElement("Einwohner")).InnerText = "80767600"
```

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:



12.4.8 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 12.51: Verwendung *GetElementsByTagName*

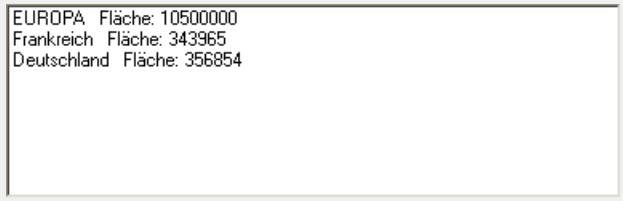
VB

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.

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
Dim list As XmlNodeList

xmldoc.Load("c:\test7.xml")
root = xmldoc.DocumentElement
ListBox1.Items.Clear()
list = xmldoc.GetElementsByTagName("Fläche")
For i As Integer = 0 To (list.Count - 1)
  ListBox1.Items.Add(list.Item(i).ParentNode.Name & "   Fläche: " &
                     list.Item(i).InnerText)
Next
```

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 12.52: Verwenden von *SelectNodes*

VB

```
Dim xmldoc As New XmlDocument()
Dim root, node As XmlNode
Dim list As XmlNodeList
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 i As Integer = 0 To (list.Count - 1)
  If list(i).InnerText.Trim <> String.Empty Then
    ListBox1.Items.Add(list(i).Name & "  (Text: " & list(i).InnerText & ")")
  Else
    ListBox1.Items.Add(list(i).Name)
  End If
Next
```

Die verschiedenen Suchstrings und das zugehörige Ergebnis:

Suchstring	Beschreibung/Ausgabe
/*	Rückgabewert ist das Root-Element: `WELT (Text: 10500000718500000543965578000003568548076760 0)`

//*	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`

Suchstring	Beschreibung/Ausgabe
	Polen NORDAMERIKA SÜDAMERIKA
/WELT	Auswahl des Root-Elements über den Namen: WELT (Text: 1050000071850000054396557800000356854807676 00)
/WELT/*	Anzeige aller Untereinträge von "WELT": AFRIKA ANTARKTIS ASIEN AUSTRALIEN EUROPA (Text: 1050000071850000054396557800000356854807676 00) NORDAMERIKA SÜDAMERIKA
/WELT/*[Frankreich]	Sucht alle Knoten unterhalb von "WELT", die einen Unterknoten mit dem Namen "Frankreich" haben: EUROPA (Text: 1050000071850000054396557800000356854807676 00)
/WELT/*[not(Frankreich)]	Sucht alle Knoten unterhalb von "WELT", die keinen Unterknoten mit dem Namen "Frankreich" haben: AFRIKA ANTARKTIS ASIEN AUSTRALIEN NORDAMERIKA SÜDAMERIKA
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
WELT/*[3]	Der vierte Untereintrag unter "WELT": ASIEN

Diese Übersicht dürfte Ihnen einen ersten Einblick in die Suchmöglichkeiten gewährt haben.

12.5 Weitere Möglichkeiten der XML-Verarbeitung

XML ist unter .NET allgegenwärtig. Wir geben hier lediglich eine kleine Auswahl der Einsatzmöglichkeiten.

12.5.1 Die relationale Sicht mit XmlDataDocument

Ausgehend von den Ausführungen des vorhergehenden Abschnitts 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 nötigen 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 12.53: Laden der Daten über das *DataSet*

VB

```
Dim ds As New DataSet()
ds.ReadXml("c:\test7.xml")
```

Hier wird das *XmlDataDocument* mit dem *DataSet* synchronisiert:

```
Dim xmlddoc As New XmlDataDocument(ds)
```

Alternativ können Sie jedoch auch erst einmal ein XSD-Schema erzeugen:

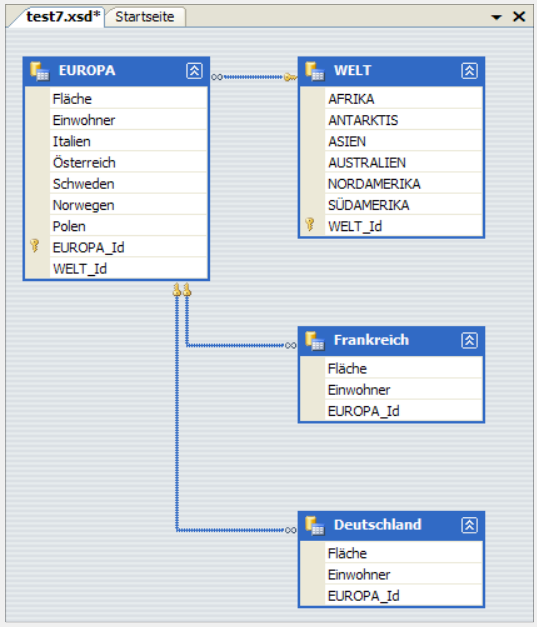
```
Dim ds As New DataSet()
ds.ReadXml("c:\test7.xml")
ds.WriteXmlSchema("c:\test7.xsd")
```

Warum dieser zusätzliche Aufwand? Die Antwort ist schnell gegeben wenn Sie versuchen, über das *XMLDataDocument*-Objekt ein XML-File zu laden, ohne entsprechende Schema-Informationen bereitzustellen. Der Versuch wird zwar erfolgreich sein, im *DataSet* werden Sie jedoch keine Tabelle vorfinden.

BEISPIEL 12.53: Laden der Daten über das *DataSet*

Ergebnis

Das erzeugte Schema für unser XML-File:



BEISPIEL 12.54: Laden des Schemas und nachfolgendes Laden der XML-Daten

VB

```
Dim xmlddoc As New XmlDataDocument()
xmlddoc.DataSet.ReadXmlSchema("c:\test7.xsd")
xmlddoc.Load("c:\test7.xml")
```

Verwendung des DataSets

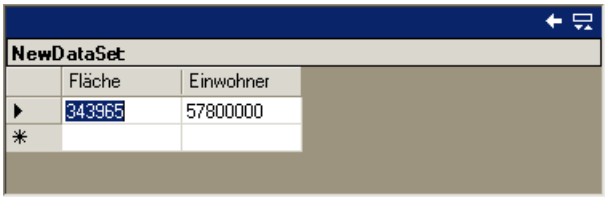
Einer Anzeige, zum Beispiel in einem *DataGrid*, steht jetzt nichts mehr im Wege:

```
Dim xmlddoc As New XmlDataDocument()
xmlddoc.DataSet.ReadXmlSchema("c:\test7.xsd")
xmlddoc.Load("c:\test7.xml")
DataGrid1.DataSource = xmlddoc.DataSet
```

HINWEIS: Wir verwenden in diesem Fall noch das "gute alte" *DataGrid*, da dieses zur Laufzeit eine Auswahl der enthaltenen *DataTables* anbietet.

Ansicht des *DataSets* im *DataGrid:*

Detailansicht (Frankreich) im *DataGrid:*



Importieren Sie die Schema-Datei in Ihr Visual Studio-Projekt, wird automatisch eine entsprechende Designer-Datei erzeugt, die das zugehörige DataSet als Klassendefinition beschreibt.

BEISPIEL 12.55: Ausschnitt aus *Test7.Designer.vb*

VB

```
...
Imports System

<System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.TypedDataSetGenerator",
 "2.0.0.0"),
 Serializable(),
 System.ComponentModel.DesignerCategoryAttribute("code"),
 System.ComponentModel.ToolboxItem(true),
 System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema"),
 System.Xml.Serialization.XmlRootAttribute("NewDataSet"),
 System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet")>
Partial Public Class NewDataSet
    Inherits System.Data.DataSet

    Private tableWELT As WELTDataTable
    Private tableEUROPA As EUROPADataTable
    Private tableFrankreich As FrankreichDataTable
    Private tableDeutschland As DeutschlandDataTable
    Private relationWELT_EUROPA As System.Data.DataRelation
    Private relationEUROPA_Frankreich As System.Data.DataRelation
    Private relationEUROPA_Deutschland As System.Data.DataRelation
...
End Class
```

12.5.2 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". VB 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 12.56: Exportieren eines einfachen Objektarrays im XML-Format

VB

Definieren des Objekts (Eigenschaften *Bezeichnung*, *Anzahl*, *Preis*):

```
Public Class Artikel
    Public Property Bezeichnung As String

    Public Property Anzahl As Integer

    Public Preis As Single
End Class
```

Erzeugen eines Objektarrays:

```
Imports System.IO
Imports System.Xml
Imports System.Xml.Serialization

Public Class Form1
...
    Private meineArtikel As New ArtikelListe()
...
        meineArtikel.art = New Artikel(3) {}
        For i As Integer = 0 To 3
            meineArtikel.art(i) = New Artikel()
        Next
        With meine Artikel
            .art(0).Anzahl = 10
            .art(0).Bezeichnung = "Mülleimer"
```

BEISPIEL 12.56: Exportieren eines einfachen Objektarrays im XML-Format

```
            .art(0).Preis = 123.45
            .art(1).Anzahl = 245
            .art(1).Bezeichnung = "Osterhasen"
            .art(1).Preis = 0.99
            .art(2).Anzahl = 44
            .art(2).Bezeichnung = "Schuhe"
            .art(2).Preis = 68.33
            .art(3).Anzahl = 2
            .art(3).Bezeichnung = "Hosen"
            .art(3).Preis = 13.45
        End With
```

Speichern im XML-Format:

```
        Dim seria As New XmlSerializer(GetType(ArtikelListe))
        Dim fs As New FileStream("c:\meineArtikel.xml", FileMode.Create)
        seria.Serialize(fs, meineArtikel)
        fs.Close()
```

Ergebnis

Die erzeugte Datei *MeineArtikel.xml* können Sie sich im Internet Explorer ansehen:

```
<?xml version="1.0" ?>
- <ArrayOfArtikel xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <Artikel>
      <Preis>123.45</Preis>
      <Bezeichnung>Mülleimer</Bezeichnung>
      <Anzahl>10</Anzahl>
    </Artikel>
  - <Artikel>
      <Preis>0.99</Preis>
      <Bezeichnung>Osterhasen</Bezeichnung>
      <Anzahl>245</Anzahl>
    </Artikel>
  - <Artikel>
      <Preis>68.33</Preis>
      <Bezeichnung>Schuhe</Bezeichnung>
      <Anzahl>44</Anzahl>
    </Artikel>
  - <Artikel>
      <Preis>13.45</Preis>
      <Bezeichnung>Hosen</Bezeichnung>
      <Anzahl>2</Anzahl>
    </Artikel>
</ArrayOfArtikel>
```

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 finden Sie in der folgenden Tabelle.

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 12.57: Eine paar Änderungen an unserer Klassendefinition zeigen die Auswirkung obiger Attribute:

VB

```
Imports System.Xml.Serialization

Public Class Artikel
    Private _bezeichnung As String
```

Eine neue Bezeichnung festlegen und als Attribut speichern:

```
    <XmlAttribute("Name")>
    Public Property Bezeichnung() As String
        Get
            Return _bezeichnung
        End Get
        Set(value As String)
            _bezeichnung = value
        End Set
    End Property
```

Eine neue Bezeichnung festlegen und als Attribut speichern:

```
    Private _Anzahl As Integer
    <XmlAttribute("Menge")>
    Public Property Anzahl() As Integer
        Get
            Return _anzahl
        End Get
        Set(value As Integer)
            _anzahl = value
        End Set
    End Property
```

BEISPIEL 12.57: Eine paar Änderungen an unserer Klassendefinition zeigen die Auswirkung obiger Attribute:

Nicht in die XML-Daten mit aufnehmen:

```
    <XmlIgnore()>
    Public Preis As Single
End Class
```

Eine übergeordnete Klasse definieren, so haben wir auch Einfluss auf den Namen der Root:

```
<XmlRoot("Warenbestand")>
Public Class ArtikelListe
    <XmlElement("Artikel")>
    Public art As Artikel()
End Class
```

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>
```

12.5.3 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, 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 12.58: Auswahl aller Orte in einer XML-Datei und Ausgabe in einer *ListBox*

VB

```
...
Imports System.Xml
Imports System.Xml.XPath
...
Public Class Form1
```

Dokument laden:

```
    Private xdoc As XPathDocument = New System.Xml.XPath.XPathDocument("Telefon.xml")
    Private xnav As XPathNavigator


    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Der *XPathNavigator* wird über ein *XPathDocument* erzeugt:

```
        xnav = xdoc.CreateNavigator()
    End Sub

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Clear()
        Dim von As Double = System.Environment.TickCount
        Try
```

Daten selektieren:

```
            Dim xit As XPathNodeIterator = 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())
            End While
```

Falls ein fehlerhafter XPath-Ausdruck eingegeben wurde:

```
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
    End Sub
```

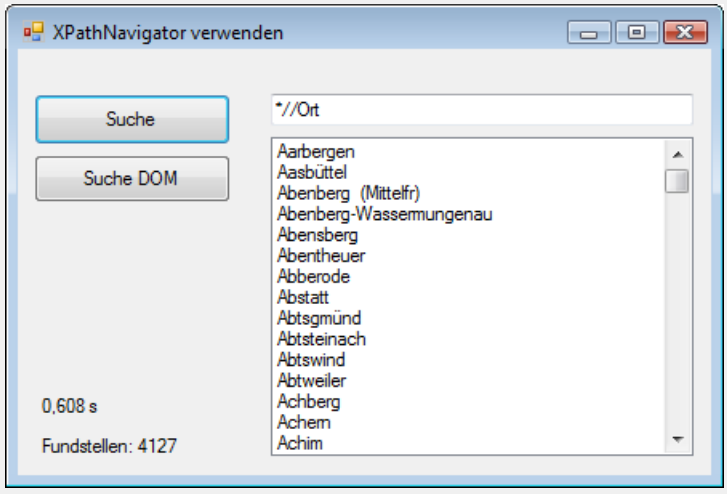
BEISPIEL 12.58: Auswahl aller Orte in einer XML-Datei und Ausgabe in einer *ListBox*

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 folgenden XPath-Ausdruck verwenden: "*//Vorwahl".

12.5.4 Schnelles Auslesen von XML-Daten mit XmlReader

Für den schnellen Lesezugriff auf XML-Dokumente bietet sich die Verwendung eines *XmlReaders* 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 wenig speicherbelastenden Alternative belohnt.

BEISPIEL 12.59: 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.

VB

```
...
Imports System.Xml
Imports System.Xml.XPath

Public Class Form1
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        ListBox1.Items.Clear()
```

Einstellungen für den *XmlReader* definieren:

```
        Dim myset As New XmlReaderSettings()
```

Leerzeichen ignorieren (fast immer angebracht):

```
        myset.IgnoreWhitespace = True
```

XmlReader erzeugen (die Daten werden **nicht** in den Speicher geladen):

```
        Dim xr As XmlReader = XmlReader.Create("Telefon.xml", myset)
```

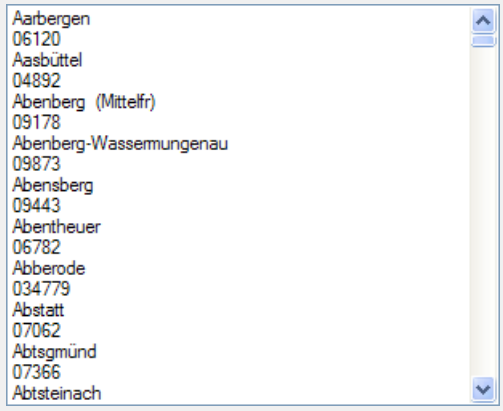
Daten durchlaufen (nur Ebene 3 anzeigen):

```
        While xr.Read()
            If xr.Depth = 3 Then
                ListBox1.Items.Add(xr.Value)
            End If
        End While

    End Sub
```

BEISPIEL 12.59: 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, 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: Der *XmlReader* verfügt auch über diverse Methoden (z.B. *ReadContentAsFloat, ReadContentAsDataTime*) zum direkten Einlesen von Datentypen. Damit bleibt Ihnen das nervtötende nachträgliche Konvertieren erspart.

12.5.5 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 12.60: ***XmlWriter***

VB

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.

```
Imports System.Xml
Imports System.Xml.XPath
```

Objekt erzeugen:

```
        Dim xw As XmlWriter.Create("Export.xml", xws)
```

Prolog erzeugen:

```
        xw.WriteStartDocument()
```

Root-Element schreiben:

```
        xw.WriteStartElement("Telefonnummern")
```

Einzel-Elemente auflisten:

```
        For i As Integer = 0 To ListBox1.Items.Count - 1 Step 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()
        Next
```

BEISPIEL 12.60: ***XmlWriter***

```
        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" />
```

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 nötig, 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.

```
...
        Dim xws As New XmlWriterSettings()
        xws.Indent = True        ' Einrücken
        xws.IndentChars = "  "
        Dim xw As XmlWriter = 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" />
```

12.5.6 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 12.61: Auslesen der Datei *Telefon.xml* und konvertieren in das XML-Format

VB

```
Imports System.Xml
Imports System.Xml.Xsl

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

        Dim xslt As New XslCompiledTransform()
        xslt.Load("Telefon.xsl")
        xslt.Transform("Telefon.xml", "Telefon.html")
    End Sub

End Class
```

Wer dachte, jetzt jede Menge Code zu produzieren, ist sicher enttäuscht worden. Die Hauptarbeit übernimmt die Transformationsdatei, die alle erforderlichen Steueranweisungen etc. enthält.

BEISPIEL 12.62: 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 12.62: 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

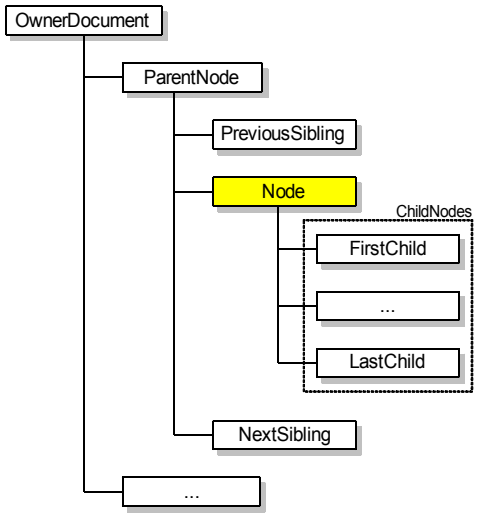
Die erzeugte HTML-Datei im Internet Explorer:

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

12.6 Praxisbeispiele

12.6.1 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 folgende Skizze (Ausgangspunkt ist der hervorgehobene Knoten).

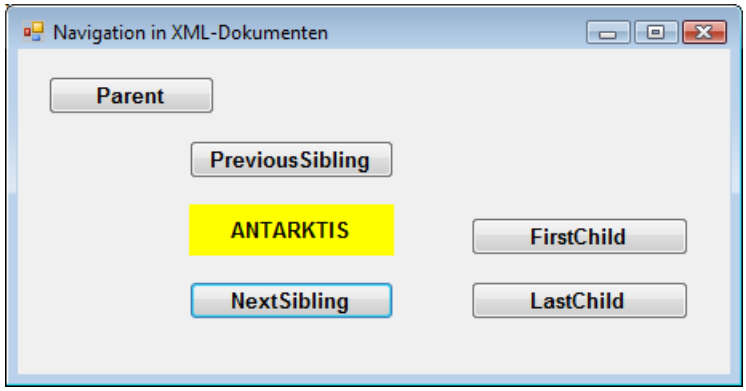


Wie Sie sehen, können Sie von jedem beliebigen Knoten aus auf den gesamten Baum zugreifen. Entweder Sie bewegen sich mit *PreviousSibling* bzw. *NextSibling* innerhalb einer Ebene, oder Sie wechseln mit *ParentNode* in die übergeordnete Ebene, um dort ebenfalls mit *PreviousSibling* bzw. *NextSibling* 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 der folgenden Abbildung (Laufzeitansicht).



Quelltext

```
Imports System.Xml

Public Class Form1
```

Zunächst definieren wir einige globale Variablen:

```
  Private xmlDoc As New XmlDocument()
  Private root As XmlElement
  Private node As XmlNode
```

Mit dem Öffnen des Fensters öffnen wir auch die Datei und weisen der Variablen *Node* den Root-Knoten zu:

```
  Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    Try
      xmlDoc.Load("test7.xml")
    Catch
      MessageBox.Show("Datei nicht gefunden!")
      Exit Sub
    End Try
    root = xmlDoc.DocumentElement
    node = root
    Label1.Text = node.Name
  End Sub
```

Wir wechseln zum Parent:

```
  Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    If Not (node.ParentNode Is Nothing) Then
      node = node.ParentNode
      Label1.Text = node.Name
    Else
      MessageBox.Show("Kein Parent vorhanden!")
    End If
  End Sub
```

Wir wechseln zum Vorgänger in der gleichen Baumebene:

```
  Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If Not (node.PreviousSibling Is Nothing) Then
      node = node.PreviousSibling
      Label1.Text = node.Name
    Else
      MessageBox.Show("Kein Vorgänger vorhanden!")
    End If
  End Sub
```

Wir wechseln zum Nachfolger in der gleichen Baumebene:

```
  Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    If Not (node.NextSibling Is Nothing) Then
      node = node.NextSibling
      Label1.Text = node.Name
    Else
      MessageBox.Show("Kein Nachfolger vorhanden!")
    End If
  End Sub
```

Wir wechseln zum ersten Child-Knoten:

```
  Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
    If node.ChildNodes.Count > 0 Then
      node = node.FirstChild
      Label1.Text = node.Name
    Else
      MessageBox.Show("Keine Untereinträge vorhanden!")
    End If
  End Sub
```

Wir wechseln zum letzten Child-Knoten:

```
  Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
    If node.ChildNodes.Count > 0 Then
      node = node.LastChild
      Label1.Text = node.Name
    Else
      MessageBox.Show("Keine Untereinträge vorhanden!")
    End If
  End Sub
End Class
```

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 *Is Nothing* oder mit *Try-Catch* können Sie zusätzliche Fehlerprüfungen einführen.

12.6.2 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

Mit dem Klick auf die Schaltfläche wird eine Instanz des *XMLDocument*-Objektes erzeugt und die Datei *daten.xml* geladen:

```
Imports System.Xml
...
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim xmlDoc As XmlDocument = New XmlDocument()
        Try
            xmlDoc.Load("Daten.xml")
        Catch
            MessageBox.Show("Datei nicht gefunden!")
            Return
        End Try
```

Gleichzeitig löschen wir die bisherigen Inhalte der *TreeView*-Komponente:

```
        TreeView1.Nodes.Clear()
        ShowNode(Nothing, xmlDoc)
        TreeView1.Nodes(0).ExpandAll()
    End Sub
```

Die Prozedur *ShowNode* rufen wir mit folgenden Argumenten auf:

- Vorgängerknoten im Baum (*Nothing*, d h. keiner) und
- Wurzelknoten der XML-Daten (*xmlDOC*):

HINWEIS: Übergeben Sie statt *xmlDOC* das Objekt *xmlDOC.DocumentElement,* beginnt die Baumansicht direkt mit dem ersten Knoten der XML-Daten, d.h. "WELT", andernfalls mit der Datei selbst.

```
    Private Sub ShowNode(parent As TreeNode, node As XmlNode)
        Dim Caption As String = String.Empty
        Dim tn As TreeNode
```

Wird kein gültiges DOM-Objekt übergeben, wird die Routine beendet:

```
        If node Is Nothing Then Return
```

Die Beschriftung des Baumknotens festlegen:

```
        If node.NodeType = XmlNodeType.Document Then Caption = "XML-Datei"
        If node.NodeType = XmlNodeType.Element Then Caption = node.Name
        If (node.NodeType = XmlNodeType.CDATA) OrElse (node.NodeType = XmlNodeType.Text) Then
            Caption = node.Value
        End If
        If Caption = "" Then Return
```

Je nach Knotentyp müssen wir andere Eigenschaften zur Bestimmung der Beschriftung auslesen.

Erzeugen eines neuen Baumknotens mit der gewählten Beschriftung:

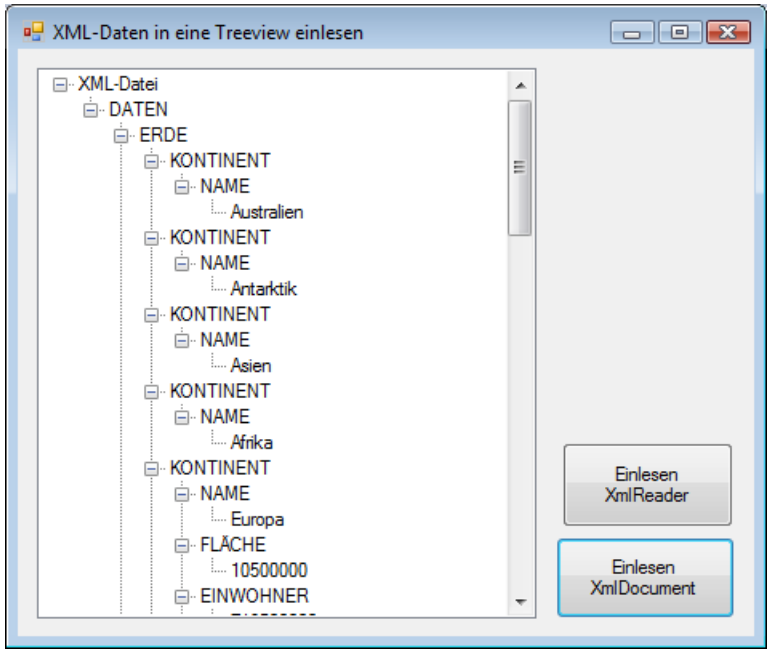
```
        If parent Is Nothing Then
            tn = TreeView1.Nodes.Add(Caption)
        Else
            tn = parent.Nodes.Add(Caption)
        End If
```

Sollten Unterelemente vorhanden sein, rufen wir für jedes dieser Elemente die aktuelle Prozedur rekursiv auf:

```
        If node.ChildNodes IsNot Nothing Then
            For i As Integer = 0 To node.ChildNodes.Count - 1
                ShowNode(tn, node.ChildNodes.Item(i))
            Next
        End If
    End Sub
```

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 Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    TreeView1.Nodes.Clear()
    ShowNode2(Nothing)
    TreeView1.Nodes(0).ExpandAll()
End Sub
```

Die eigentliche Routine zum Einlesen:

```
Private Sub ShowNode2(parent As TreeNode)
    Dim tn As TreeNode = parent
    Dim myset As New XmlReaderSettings()
    myset.IgnoreWhitespace = True
```

Da unsere XML-Datei eine DTD besitzt, müssen wir deren Verarbeitung vorher zustimmen:

```
    myset.ProhibitDtd = False
```

Den *XmlReader* initalisieren:

```
    Dim xr As XmlReader = 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 Then
            tn = tn.Parent
        Else
```

Elemente besitzen einen *Name*:

```
            If xr.NodeType = XmlNodeType.Element Then
                If tn Is Nothing Then
                    tn = TreeView1.Nodes.Add(xr.Name)
                Else
                    tn = tn.Nodes.Add(xr.Name)
                End If
            End If
```

Texte besitzen einen *Value*:

```
            If xr.NodeType = XmlNodeType.Text Then tn.Text = tn.Text & ": " & xr.Value
        End If
    End While
End Sub
```

12.6.3 DataSets in XML-Strings konvertieren

Dieses Beispiel demonstriert Ihnen, wie Sie ein beliebiges *DataSet*-Objekt in einen XML-String umwandeln, gleichzeitig wird auch die Rücktransformation aufgezeigt.

Ganz nebenbei erfahren Sie auch, wie Sie ein *DataSet* (inklusive Relationen zwischen den Tabellen) zur Laufzeit erstellen und füllen können.

Oberfläche

Ein Windows *Form*, eine *TextBox* (*MultiLine* = *True*), zwei Schaltflächen und das "gute alte" *DataGrid* bilden die Testoberfläche.

HINWEIS: Im Gegensatz zum neuen *DataGridView* kann das *DataGrid* auch mehrere Tabellen und ihre Beziehungen gleichzeitig darstellen.

Quelltext

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.

```
Imports System.Text
Imports System.Xml
Imports System.IO
Public Class Form1
```

Die folgende Methode konvertiert das übergebene *DataSet* in einen XML-String, wobei der Weg über einen *MemoryStream* und ein *Byte*-Array geht:

```
    Public Function ConvertDataSetToXML(ds As DataSet) As String
        Dim stream As MemoryStream = Nothing
        Dim writer As XmlTextWriter = Nothing
        Try
            stream = New MemoryStream()
```

XmlTextWriter mit dem *MemoryStream* initialisieren:

```
            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):

```
            Dim count As Integer = stream.Length
            Dim arr(count-1) As Byte
```

XML-String aus Byte-Array gewinnen und zurückgeben:

```
            stream.Seek(0, SeekOrigin.Begin)
            stream.Read(arr, 0, count)
            Dim utf As New UnicodeEncoding()
            Return utf.GetString(arr).Trim()
        Catch
            Return String.Empty
        Finally
            If writer IsNot Nothing Then writer.Close()
        End Try
    End Function
```

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 Function ConvertXMLToDataSet(xml As String) As DataSet
        Dim reader As StringReader = Nothing
        Try
            Dim ds As New DataSet()
            reader = New StringReader(xml)
```

XML-String in *DataSet* einlesen:

```
            ds.ReadXml(reader)
            Return ds
        Catch
            Return Nothing
        Finally
            If reader IsNot Nothing Then reader.Close()
        End Try
    End Function
```

Die folgende Methode *getTestDS* erzeugt ein untypisiertes *DataSet* mit zwei *DataTable*s und einer *DataRelation* und fügt jeder Tabelle zwei Datensätze hinzu:

```
    Private Shared Function getTestDS() As DataSet
```

Tabelle *Personen*:

```
        Dim dt1 As New DataTable("Personen")
```

Primärschlüssel:

```
        Dim col1 As DataColumn = dt1.Columns.Add("Nr", GetType(Integer))
        col1.AllowDBNull = False
        col1.Unique = True
        col1.AutoIncrement = True
        col1.AutoIncrementStep = 1
```

Die restlichen Spalten hinzufügen:

```
        dt1.Columns.Add("Vorname", GetType(String))
        dt1.Columns.Add("Nachname", GetType(String))
```

```
dt1.Columns.Add("Geburtstag", GetType(Date))
```

Zwei Datensätze hinzufügen:

```
Dim rw11 As DataRow = dt1.NewRow()
rw11("Vorname") = "Klaus"
rw11("Nachname") = "Mller"
rw11("Geburtstag") = Convert.ToDateTime("3.4.1975")

Dim rw12 As DataRow = 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*:

```
Dim dt2 As New DataTable("Bestellungen")
' Primärschlüssel:
Dim col2 As DataColumn = dt2.Columns.Add("Nr", GetType(Integer))
col2.AllowDBNull = False
col2.Unique = True
col2.AutoIncrement = True
col2.AutoIncrementStep = 1
dt2.Columns.Add("Datum", GetType(Date))
dt2.Columns.Add("Betrag", GetType(Decimal))
dt2.Columns.Add("PersNr", GetType(Integer))
' Fremdschlüssel
dt2.Columns.Add("Bemerkung", GetType(String))
```

Zwei Datensätze hinzufügen:

```
Dim rw21 As DataRow = dt2.NewRow()
rw21("Datum") = Convert.ToDateTime("20.2.2006")
rw21("Betrag") = Convert.ToDecimal("256,50")
rw21("PersNr") = 0
rw21("Bemerkung") = "per Nachname"
dt2.Rows.Add(rw21)

Dim rw22 As DataRow = dt2.NewRow()
rw22("Datum") = Convert.ToDateTime("8.3.2006")
rw22("Betrag") = Convert.ToDecimal("12,95")
rw22("PersNr") = 0
rw22("Bemerkung") = ("per Scheck")
dt2.Rows.Add(rw22)
```

DataSet zusammenbauen (mit 1:n-Relation zwischen *Kunden* und *Bestellungen*):

```
Dim ds As New DataSet()
ds.Tables.Add(dt1)
ds.Tables.Add(dt2)
```

```
        ds.Relations.Add("Person_Bestellungen", ds.Tables("Personen").Columns("Nr"),
                                                ds.Tables("Bestellungen").Columns("PersNr"))
        Return ds
    End Function
```

DataSet in XML-String:

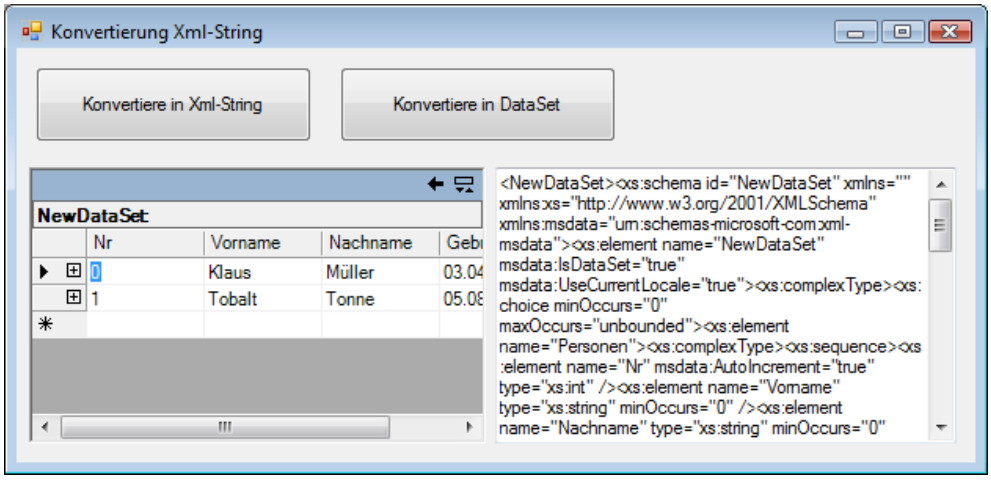
```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim ds As DataSet = getTestDS()
        TextBox1.Text = ConvertDataSetToXML(ds)
        Button2.Enabled = True
    End Sub
```

XML-String in *DataSet*:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim ds As DataSet = ConvertXMLToDataSet(TextBox1.Text)
        DataGrid1.DataSource = Nothing
        DataGrid1.DataSource = ds
    End Sub

End Class
```

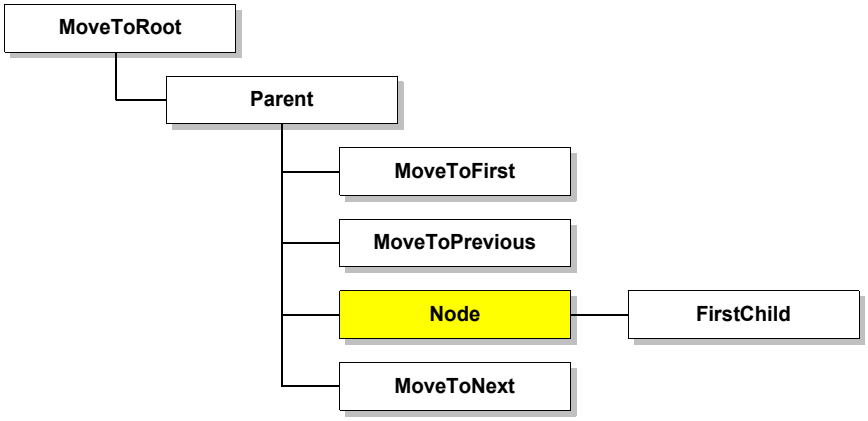
Test

Zunächst lassen wir uns die XML-Darstellung des *DataSet*s rechts in der *TextBox* anzeigen. Anschließend betätigen wir zwecks Rückkonvertierung die Schaltfläche *Konvertiere in DataSet*:



12.6.4 In Dokumenten mit dem XPathNavigator navigieren

Welche Möglichkeiten, d h. Methoden und Eigenschaften, das *XPathNavigator*-Objekt zur Navigation zwischen den einzelnen Knoten anbietet, zeigt die folgende Skizze (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 *MoveToPrevious* bzw. *MoveToNext* innerhalb einer Ebene oder Sie wechseln mit *Parent* in die übergeordnete Ebene, um dort ebenfalls mit *MoveToPrevoius* bzw. *MoveToNext* 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 *MoveToPrevious* bzw. *MoveToNext* mit den weiteren Elementen der dann aktiven Ebene fortzufahren.

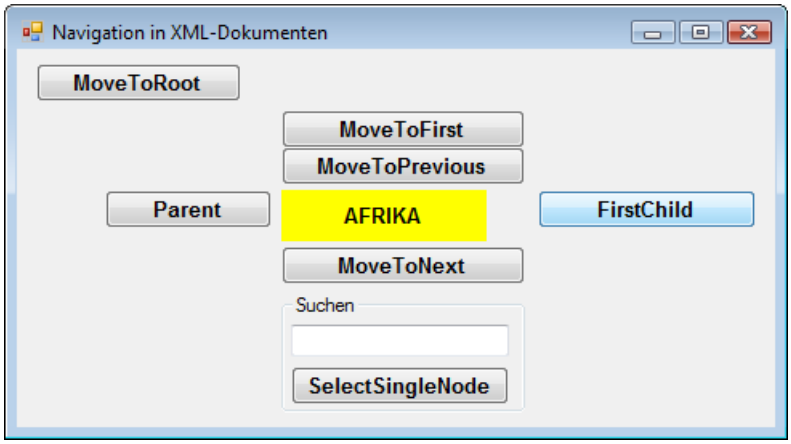
HINWEIS: Die Root des XML-Baums erreichen Sie in jedem Fall mit *MoveToRoot*. Das per Definition immer eine Root vorhanden sein muss, gibt diese Methode auch kein *True* oder *False* zurück.

Unser Beispiel 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 Tasten die jeweiligen Methoden zur Verfügung. Ist der Knoten gewechselt, wird die Bezeichnung angezeigt.

Zusätzlich finden Sie im Programm auch eine Unterstützung für die Methode *SelectSingleNode*, mit der sich ein einzelnen Knoten suchen lässt.

Oberfläche

Entwerfen Sie eine Oberfläche nach folgendem Vorbild (Laufzeitansicht):



Quelltext

```
Imports System.Xml
Imports System.Xml.XPath

Public Class Form1
    Private xmlDoc As New System.Xml.XmlDocument()
    Private nav As XPathNavigator
```

Mit dem Öffnen des Fensters öffnen wir auch die Datei und erzeugen unseren *XPathNavigator*:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Try
            xmlDoc.Load("test7.xml")
            nav = xmlDoc.CreateNavigator()
            Label1.Text = nav.Name
        Catch
            MessageBox.Show("Datei nicht gefunden!")
        End Try
    End Sub
```

Wir wechseln zum Vorgänger

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        If nav.MoveToPrevious() Then
            Label1.Text = nav.Name
        Else
            MessageBox.Show("Kein Vorgnger vorhanden!")
        End If
    End Sub
```

Wir wechseln zum Nachfolger:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        If nav.MoveToNext() Then
            Label1.Text = nav.Name
```

```
        Else
            MessageBox.Show("Kein Nachfolger vorhanden!")
        End If
    End Sub
```

Wir wechseln zu den Untereinträgen:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        If nav.MoveToFirstChild() Then
            Label1.Text = nav.Name
        Else
            MessageBox.Show("Keine Untereintrge vorhanden!")
        End If
    End Sub
```

Wir wechseln zum Parent:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        If nav.MoveToParent() Then
            Label1.Text = nav.Name
        Else
            MessageBox.Show("Kein Parent vorhanden!")
        End If
    End Sub
```

Wir wechseln zum ersten Knoten der gleichen Ebene:

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        If nav.MoveToFirst() Then
            Label1.Text = nav.Name
        End If
    End Sub
```

Wir wechseln zur Root:

```
    Private Sub Button7_Click(sender As Object, e As EventArgs) Handles Button7.Click
        nav.MoveToRoot()
        Label1.Text = nav.Name
    End Sub
```

Wir suchen in den Baumzweigen:

```
    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        Try
            Dim nav2 As XPathNavigator = nav.SelectSingleNode(TextBox1.Text)
            If nav2 IsNot Nothing Then
                nav = nav2
                Label1.Text = nav.Name
            Else
                MessageBox.Show("Nicht gefunden!")
            End If
```

Hier sollten wir auf eine Fehlerbehandlung nicht verzichten, falls ein fehlerhafter XPath-Ausdruck eingegeben wird:

```
        Catch generatedExceptionName As Exception
            MessageBox.Show("Fehlerhafter Ausdruck!")
        End Try
    End Sub

End Class
```

Test

Starten Sie das Programm und versuchen Sie, durch den XML-Baum zu navigieren. Der Inhalt der XML-Datei soll Ihnen dabei als Hilfestellung dienen (siehe folgende Abbildung).

HINWEIS: Beachten Sie, dass sich auch über "WELT" noch ein Objekt befindet!

Bemerkung

Haben Sie bisher mit dem DOM (*Document Object Model*) gearbeitet, wird Ihnen sicher nicht verborgen geblieben sein, dass der Quellcode wesentlich lesbarer geworden ist. Auch sinnlose Abfragen auf vorhandene Knoten sind nicht mehr nötig.

Kapitel 13

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 ADO.NET, wobei der Schwerpunkt auf dem ersten Teil des Objektmodells, den Datenprovidern, liegt.

Erst das nachfolgende Kapitel 14 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 erst in den spezielleren Kapiteln 26 und 33 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, einen kleinen Vorgeschmack bieten das Praxisbeispiel 14.6.5.

13.1 Eine kleine Übersicht

Die umfangreichen Klassenbibliotheken von ADO.NET verlangen vom Einsteiger eine ziemlich steile "Lernkurve".Er tut deshalb gut daran, sich zunächst einen Gesamtüberblick zu verschaffen.

13.1.1 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.

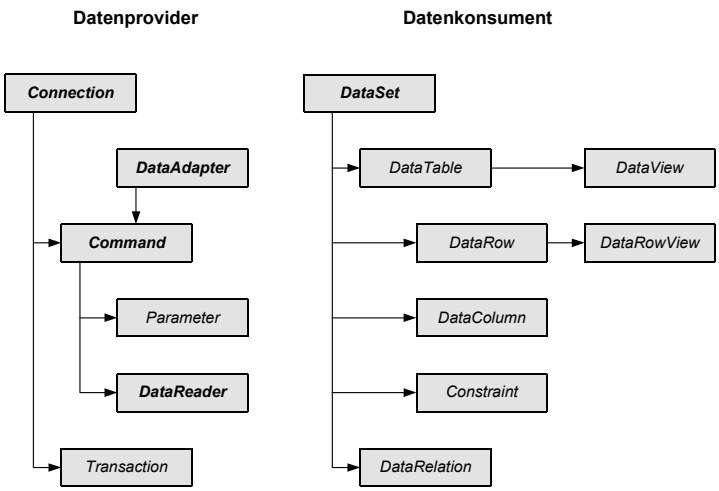
Die Objekte *Connection*, *Command*, *DataReader* und *DataAdapter* sind die Hauptelemente des .NET-Datenprovider-Modells.

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).

Um ein erstes praktisches Feeling für die ADO.Net-Klassen zu entwickeln, sollte der Einsteiger bereits jetzt ein einfaches Beispiel ausprobieren, z.B. das Praxisbeispiel

► 13.8.1 Wichtige ADO.NET-Objekte im Einsatz



13.1.2 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

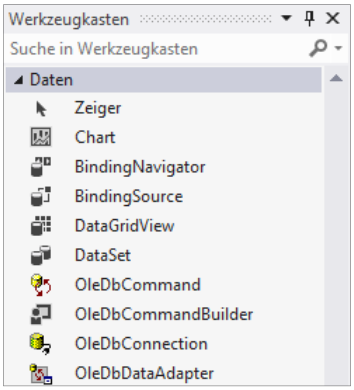
Weitere Datenprovider

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 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...* (oder direkt das Menü *Extras/Werkzeugkastenelemente auswählen...*) und fügen Sie die gewünschten Komponenten hinzu:



Anzeige der installierten Datenprovider

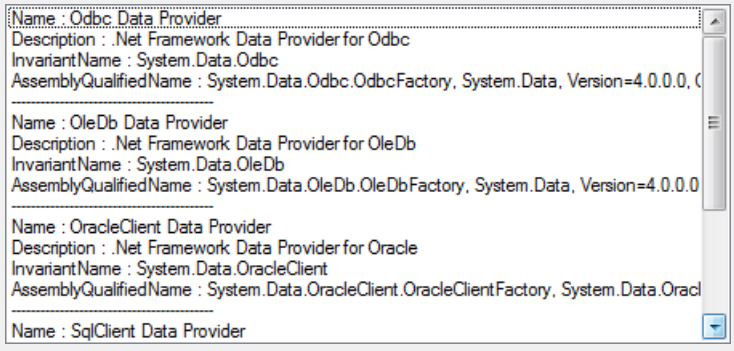
Einen Überblick über alle auf Ihrem System installierten ADO.NET-Datenprovider können Sie mit der Methode *GetFactoryClasses* der *DbProviderFactories*-Auflistung aus dem *System.Data.Common*-Namespace gewinnen.

BEISPIEL 13.1: Alle verfügbaren Datenprovider in einer *ListBox* anzeigen.

VB

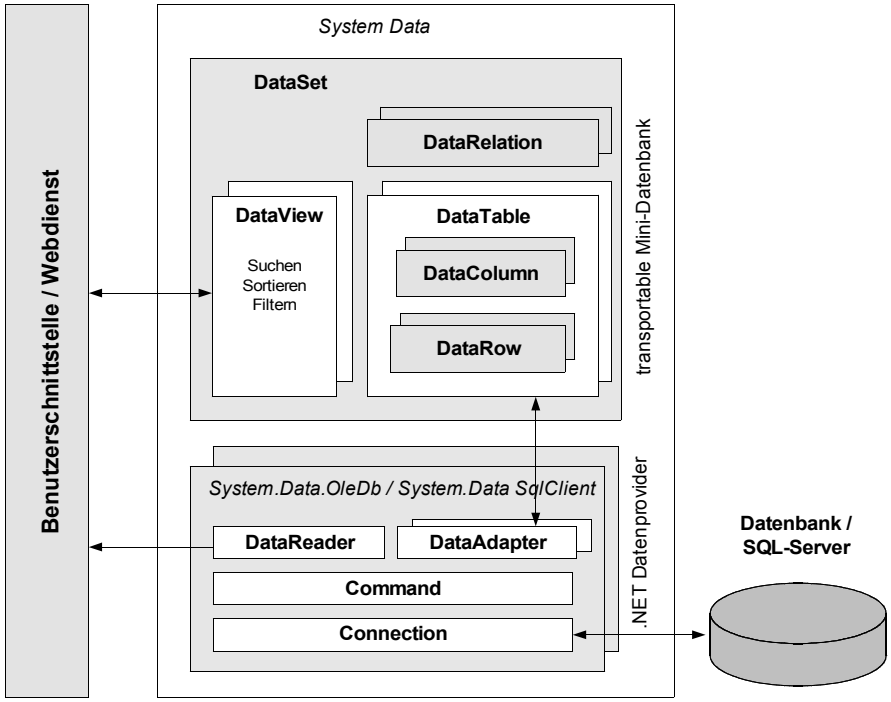
```
Dim providers As DataTable = System.Data.Common.DbProviderFactories.GetFactoryClasses()
For Each provider As DataRow In providers.Rows
    For Each col As DataColumn In providers.Columns
        ListBox1.Items.Add(col.ColumnName & " : " & provider(col))
    Next
    ListBox1.Items.Add("--------------------------------------")
Next
```

Ergebnis



13.1.3 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.

13.2 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.

13.2.1 Allgemeiner Aufbau

Der am häufigsten zum Erzeugen und Initialisieren eines *Connection*-Objekts benutzte Konstruktor nimmt einen *ConnectionString* als Parameter entgegen:

SYNTAX: `Dim conn As New Connection(ConnectionString As String)`

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 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.

13.2.2 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 Registry 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 13.2: Öffnen einer OLE DB-Verbindung zur Access-Datenbank *Nordwind.mdb*, die sich im aktuellen Anwendungsverzeichnis befindet.

VB
```
Imports System.Data.OleDb
...
Dim conn As 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 13.3: Eine zum Vorgängerbeispiel äquivalente Variante

VB
```
Imports System.Data.OleDb
...
Dim conn As 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.

13.2.3 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 13.4: 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.

VB
```
Imports System.Data.OleDb
...
Dim conn As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Firma.mdb;")
```

Verbindung öffnen:

```
conn.Open()
```

Daten übertragen:

```
Dim ds As New DataSet()
Dim da As 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 13.3.5 diskutiert.

13.2.4 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 13.5: Zwei gleichwertige Möglichkeiten.

VB
```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=(LocalDB)\v11.0; " &
                                        "Initial Catalog=Northwind;Integrated Security=SSPI"
```

oder

```
conn.ConnectionString = "Provider=SQLOLEDB.1; Data Source=(LocalDB)\v11.0; " &
                                        "Database=Northwind;Integrated Security=SSPI"
Label1.Text = conn.DataSource                ' liefert "(LocalDB)\v11.0"
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).

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 13.6: Die Zeit bis zum Timeout wird auf 10 Sekunden festgelegt.

VB
```
conn.ConnectionString =
          "Provider=SQLOLEDB.1; Data Source=(LocalDB)\v11.0;..; 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

13.2.5 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 13.7: Zwei äquivalente Varianten zum Wechseln der Datenbank.

VB

```
conn.ChangeDatabase("Northwind")
```

oder

```
Dim cmd As OleDbCommand = 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).

13.2.6 Der ConnectionStringBuilder

Um das Zusammenbauen eines *ConnectionString*s etwas übersichtlicher zu gestalten, gibt es die providerspezifische *ConnectionStringBuilder*-Klasse.

BEISPIEL 13.8: Vergleich von zwei Möglichkeiten für das Erstellen einer Verbindungszeichenfolge zur *Northwind*-Datenbank des SQL Servers

VB

```
Imports System.Data.SqlClient
```

Ohne *ConnectionStringBuilder*:

```
Dim connStr As String =
        "Data Source = .\SQLEXPRESS; Initial Catalog=Northwind; Integrated Security=True"
conn = New SqlConnection(connStr)
```

Mit *ConnectionStringBuilder*:

```
Dim csb As New SqlConnectionStringBuilder()

csb.DataSource = ".\SQLEXPRESS"
csb.IntegratedSecurity = True
csb.InitialCatalog = "Northwind"
```

13.3 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.

13.3.1 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: `Dim cmd As New Command(sql As String, conn As Connection)`

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 13.9: Erzeugen und Anwenden eines *Command*-Objekts

VB

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.

```
Dim conn As New OleDbConnection(
                "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
Dim updCmd As New OleDbCommand(
                "UPDATE Kunden SET Firma = 'Pariser Kunde' WHERE Ort = 'Paris'", conn)
Dim selCmd As New OleDbCommand(
                "SELECT Firma, Kontaktperson, Ort FROM Kunden WHERE Ort = 'Paris'", conn)
Dim da As New OleDbDataAdapter(selCmd)
Dim ds As New DataSet()
conn.Open()
```

UPDATE-Befehl wird gegen die Datenbank ausgeführt:

```
updCmd.ExecuteNonQuery()
```

DataSet erhält neue Tabelle ("PariserKunden") mit Datensätzen gemäß SELECT-Befehl:

```
da.Fill(ds, "PariserKunden")
conn.Close()
```

Den vollständigen Code finden Sie im Praxisbeispiel

► 13.8.2 Eine Aktionsabfrage ausführen

13.3.2 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 13.10: Zwei äquivalente Varianten, wenn ein *Connection*-Objekt *conn* vorliegt.

VB

Variante A:

```
Dim cmd As New OleDbCommand()
cmd.Connection = conn
```

Variante B:

```
Dim cmd As OleDbCommand = conn.CreateCommand()
```

13.3.3 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 13.11: Zwei Varianten zum Erzeugen und Initialisieren eines *OleDbCommand*-Objekts

VB

```
Dim cmd As New OleDbCommand(
                    "UPDATE Kunden SET Firma = 'Pariser Firma' WHERE Ort = 'Paris'", conn)
```

... ist äquivalent zu:

```
Dim cmd As 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 13.12: Ein *DataSet* wird mit der Kundenliste der SQL-Datenbank *Northwind* gefüllt, wofür maximal 10 Sekunden zur Verfügung stehen.

VB

```
Dim conn As New SqlConnection(
                  "Data Source=(local);Integrated Security=sspi;Initial Catalog=Northwind")
Dim cmd As New SqlCommand("SELECT CustomerID, CompanyName FROM Customers", conn)
```

Die Ausführung der Abfrage darf maximal 30 sek dauern:

```
cmd.CommandTimeout = 30
```

BEISPIEL 13.12: Ein *DataSet* wird mit der Kundenliste der SQL-Datenbank *Northwind* gefüllt, wofür maximal 10 Sekunden zur Verfügung stehen.

```
Dim da As New SqlDataAdapter()
da.SelectCommand = cmd
Dim ds As 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 13.13: Eine gleichwertige Realisierung des Vorgängerbeispiels

VB
```
Dim da As 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 13.14: Aufruf der Stored Procedure "Sales by Years" in der Datenbank *Northwind*

VB
```
Dim cmd As New SqlCommand("Sales by Year", conn)
cmd.CommandType = CommandType.StoredProcedure
Dim parm1 As New SqlParameter("@Beginning_Date", SqlDbType.DateTime)
```

Definition als Input-Parameter:

```
parm1.Direction = ParameterDirection.Input
```

BEISPIEL 13.14: Aufruf der Stored Procedure "Sales by Years" in der Datenbank *Northwind*

Das *Beginn*-Datum wird der ersten *TextBox* entnommen:

```
parm1.Value = Convert.ToDateTime(TextBox1.Text)
```

Parameter hinzufügen:

```
cmd.Parameters.Add(parm1)
```

Der nach dem gleichen Prinzip funktionierende Aufruf einer in der Datenbank *Nordwind.mdb* gespeicherten Auswahlabfrage ist im folgenden Praxisbeispiel erklärt:

► 13.8.3 Eine Auswahlabfrage aufrufen

13.3.4 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 13.15: *ExecuteNonQuery*-Methode

VB

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).

```
Dim updStr As String = "UPDATE Kunden SET Firma = 'Pariser Firma' WHERE Ort = 'Paris'"
Dim updCmd As New OleDbCommand(updStr, conn)
conn.Open()
```

SQL-Anweisung ausführen und Anzahl betroffener Datensätze anzeigen:

```
Label1.Text = cmd.ExecuteNonQuery.ToString()
```

Das ausführliche Beispiel findet sich im Praxisbeispiel

► 13.8.3 Eine Auswahlabfrage aufrufen

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 13.16: Auf Basis eines gültigen *OleDbConnection*-Objekts und eines SELECT-Befehls wird ein *OleDbCommand*-Objekt erstellt und zum Erzeugen eines *DataReader*-Objekts verwendet.

VB
```
Dim selStr As String = "SELECT Firma, Kontaktperson, Ort FROM Kunden WHERE Ort = 'Paris'"
Dim selCmd As New OleDbCommand(selStr, conn)

conn.Open()
Dim dr As OleDbDataReader = 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 13.17: Abfrage des Namens der Firma eines bestimmten Kunden.

VB
```
Dim cmd As New SqlCommand("SELECT Firma FROM Kunden WHERE KundenCode = 'ALFKI'", conn)
conn.Open()
Label1.Text = Convert.ToString(cmd.ExecuteScalar)
```

Besonders vorteilhaft kann man *ExecuteScalar* zur Ausführung von Aggregatfunktionen verwenden, was weniger Aufwand erfordert als die Anwendung der *ExecuteReader*-Methode.

BEISPIEL 13.18: Aus der Datenbank *Northwind* wird die Anzahl der in Paris wohnhaften Kunden abgefragt und angezeigt.

VB
```
Dim cmd As 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()
```

13.3.5 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.

[1] Auch die meisten Codebeispiele dieses Buchs bilden da (aus Platzgründen!) leider keine Ausnahme.

BEISPIEL 13.19: Die (leider nicht ganz saubere) Programmierung einer Datenbankverbindung

VB
```
Dim conn As New SqlConnection(connString)
Dim cmd As 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 13.20: Saubere Programmierung des Vorgängerbeispiels

VB
```
Using conn As New SqlConnection(connString)
   Using cmd As New SqlCommand(cmdString, conn)
      conn.Open()
      cmd.ExecuteNonQuery()
   End Using
End Using
```

Der intern generierte Code für obige Zeilen dürfte etwa folgendermaßen aussehen:

```
Dim conn As SqlConnection = Nothing
Dim cmd As SqlCommand = Nothing
Try
   conn = New SqlConnection(connString)
   cmd = New SqlCommand(cmdString, conn)
   conn.Open()
   cmd.ExecuteNonQuery()
Finally
   If Nothing IsNot cmd Then cmd.Dispose()
   If Nothing IsNot conn Then conn.Dispose()
End Try
```

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.

13.4 Parameter-Objekte

In vielen Fällen enthält ein *Command*-Objekt Parameter bzw. Parameter-Auflistungen, mit denen parametrisierte Abfragen durchführbar sind.

13.4.1 Erzeugen und Anwenden eines Parameter-Objekts

Einer der möglichen Konstruktoren:

SYNTAX: `Dim prm As New Parameter(pName As String, pType As DbType)`

Nach dem Zuweisen weiterer Eigenschaften erfolgt das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
cmd.Parameters.Add(prm As Parameter)
```

BEISPIEL 13.21: Ein *SqlParameter*-Objekt *p1* wird zur *Parameters*-Collection eines vorhandenen *SqlCommand*-Objekts hinzugefügt.

VB

Im Konstruktor Namen und Datentyp übergeben:

```
Dim p1 As New SqlParameter("@Geburtstag", SqlDbType.DateTime)
```

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 13.22: Der im Vorgängerbeispiel definierte Parameter *@Geburtstag* wird in eine SQL-Abfrage eingefügt.

VB

```
cmd.CommandText = "SELECT * FROM Employees WHERE (BirthDate > @Geburtstag)"
```

13.4.2 Eigenschaften des Parameter-Objekts

ParameterName- und Value-Eigenschaft

Beide Eigenschaften dürften selbsterklärend sein.

BEISPIEL 13.23: Eine alternative Zuweisung für das obige erste Beispiel wäre:

VB

```
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 13.24: Ein *Byte*-Parameter wird erzeugt, initialisiert und zur *Parameters*-Collection eines *SqlCommand*-Objekts hinzugefügt.

VB
```
Dim prm As SqlParameter = 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 13.25: Ein *OleDbParameter* wird erstellt und seine *Direction*-Eigenschaft festgelegt.

VB
```
Public Sub CreateOleDbParameter()
    Dim p1 As New OleDbParameter("Description", OleDbType.VarChar, 50)
    p1.IsNullable = True
    p1.Direction = ParameterDirection.Output
End Sub
```

13.5 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.

13.5.1 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: `Dim cmdBuilder As New CommandBuilder(da As DataAdapter)`

Der *CommandBuilder* verfolgt nun argwöhnisch alle am *DataSet* vorgenommenen Änderungen und generiert die erforderlichen Queries bzw. *Command*-Objekte selbstständig im Hintergrund.

13.5.2 Anwenden

Die *Update*-Methode des *DataAdapter*s würde im folgenden Beispiel ohne *OleDbCommandBuilder* fehlschlagen.

BEISPIEL 13.26: Aktualisieren der Kunden-Tabelle aus *Nordwind.mdb*.

VB

Beim Instanziieren erhält der *DataAdapter* automatisch auch seine *SelectCommand*-Eigenschaft, sodass diese nicht explizit zugewiesen werden muss:

```
Dim da As New OleDbDataAdapter("SELECT * FROM Kunden", conn)
```

Ein *OleDbCommandBuilder* wird mit dem *OleDbDataAdapter* verbunden:

```
Dim cmdB As New OleDbCommandBuilder(da)
```

Ein *DataSet* wird mit den Daten gefüllt:

```
Dim ds As 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.

Ein Beispiel für den Einsatz des *CommandBuilder*s finden Sie im Praxisbeispiel

► 13.8.4 Die Datenbank aktualisieren.

13.6 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*).

13.6.1 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: `Dim myReader As DataReader = 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 13.27: Ein *DataReader*, der auch das Schließen des *Connection*-Objekts erledigt, wird instanziiert.

VB
```
Dim dr As DataReader = cmd.ExecuteReader(CommandBehavior.CloseConnection)
```

13.6.2 Daten lesen

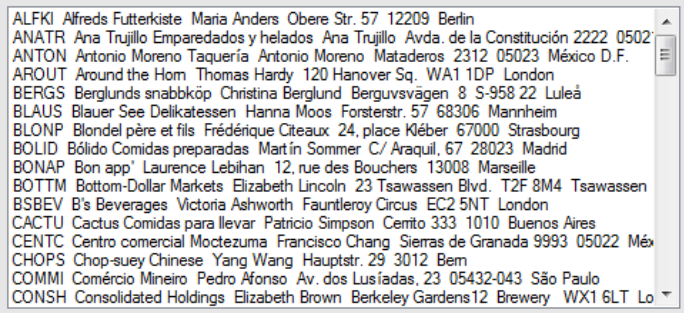
Das Auslesen der Informationen innerhalb einer Schleife ist typisch für die Arbeit mit dem *DataReader*.

BEISPIEL 13.28: Die Kundentabelle aus *Nordwind.mdb* wird zeilenweise in eine *ListBox* ausgelesen.

VB
```
Const SQL As String = "SELECT * FROM Kunden ORDER BY KundenCode"
Dim conn As New OleDbConnection(
                        "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
Dim cmd As New OleDbCommand(SQL, conn)
Dim dr As OleDbDataReader
conn.Open()
dr = cmd.ExecuteReader()
Dim str As String = String.Empty
Dim tab As String = "  "
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)
End While
dr.Close()
conn.Close()
```

BEISPIEL 13.28: Die Kundentabelle aus *Nordwind.mdb* wird zeilenweise in eine *ListBox* ausgelesen.

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!

13.6.3 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.

13.6.4 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 13.29: Ein Datumswert aus der *Employee*-Tabelle der *Northwind*-Datenbank wird ausgelesen.

VB
```
Dim aDate As DateTime
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 13.30: Einlesen von Kundendaten

VB
```
Dim kd As New CKunde()
kd.ID = Convert.ToInt64(DataReader("id"))
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.

13.7 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.

13.7.1 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: `Dim myDataAdapter As New DataAdapter(selectStr As String, conn As Connection)`

BEISPIEL 13.31: Ein *DataAdapter* füllt ein *DataSet* mit Datensätzen aus *Nordwind.mdb*.

VB
```
Imports System.Data.OleDb
...
Dim conn As New OleDbConnection(
                        "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
Dim da As New OleDbDataAdapter("SELECT * FROM Kunden WHERE Ort = 'Paris'", conn)
Dim ds As 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: `Dim myDataAdapter As New DataAdapter(selectCommand As Command)`

BEISPIEL 13.32: Das Vorgängerbeispiel wird mit einem *Command*-Objekt realisiert.

VB
```
Imports System.Data.OleDb
...
Dim conn As New OleDbConnection(
                       "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
Dim cmd As New OleDbCommand("SELECT Firma FROM Kunden WHERE Ort = 'Paris'")
cmd.Connection = conn
Dim da As New OleDbDataAdapter(cmd)
Dim ds As New DataSet()
conn.Open()
da.Fill(ds, "PariserKunden")
conn.Close()
```

13.7.2 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 13.33: Realisierung der Vorgängerbeispiele mittels *SelectCommand*-Eigenschaft

VB
```
Imports System.Data.OleDb
...
Dim conn As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Nordwind.mdb;")
Dim cmd As New OleDbCommand("SELECT Firma FROM Kunden WHERE Ort = 'Paris'")
cmd.Connection = conn
Dim da As New OleDbDataAdapter()
da.SelectCommand = cmd
Dim ds As 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.

13.7.3 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: `myDataAdapter.Fill(ds As DataSet, tblName As String)`

BEISPIEL 13.34: Ein *DataSet* wird mit der Kundentabelle aus *Nordwind.mdb* gefüllt. Im *DataSet* sollen die Namen aller Firmen aus Paris geändert werden in "Pariser Firma".

VB
```
Dim conn As New OleDbConnection(
                              "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
Dim da As New OleDbDataAdapter("SELECT * FROM Kunden", conn)
Dim ds As New DataSet()
da.Fill(ds, "Kunden")
```

Das Arbeiten mit den Daten im *DataSet*:

```
Dim dt As DataTable = ds.Tables("Kunden")
```

Alle Zeilen der *DataTable* durchlaufen:

```
For Each cRow As DataRow In dt.Rows
    If cRow("Ort").ToString() = "Paris" Then 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:
```
Dim z As Integer = myDataAdapter.Fill(ds As DataSet, start As Integer,
anzahl As Integer, tblName As String)
```

start = Nummer der Startzeile
anzahl = Anzahl der abzurufenden Datensätze
z = Anzahl der tatsächlich zurückgegebenen Datensätze

BEISPIEL 13.35: Ab Zeile 100 werden 50 Zeilen aus der Datenbank abgerufen und in die "Kunden"-Tabelle gefüllt.

VB
```
Dim z As Integer = da.Fill(ds, 100, 50, "Kunden")
```

13.7.4 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:
```
myDataAdapter.Update(ds As DataSet, tblName As String)
```

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 13.36: (*Fortsetzung*)

VB

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):

```
...
Dim cmd As 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:

```
Dim prm As OleDbParameter = cmd.Parameters.Add("@p2", OleDbType.VarChar)
prm.SourceColumn = "KundenCode"
```

BEISPIEL 13.36: (*Fortsetzung*)

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 Praxisbeispiel

► 13.8.4 Die Datenbank aktualisieren

13.8 Praxisbeispiele

13.8.1 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

```
Imports System.Data.OleDb
Public Class Form1
```

Alles passiert beim Laden des Formulars und beginnt mit dem Festlegen 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:

```
    Protected Overrides Sub OnLoad(e As EventArgs)
```

```
        Dim conn As New OleDbConnection(
                                "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb")
        Dim cmdStr As String =
           "SELECT ArtikelNr,Artikelname,Liefereinheit,Einzelpreis,Mindestbestand FROM Artikel"
        Dim cmd As New OleDbCommand(cmdStr, conn)
```

Nun geht es um das Füllen des *DataSet*s mit Hilfe des *DataAdapter*s:

```
        Dim da As New OleDbDataAdapter(cmd)
        Dim ds As 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:

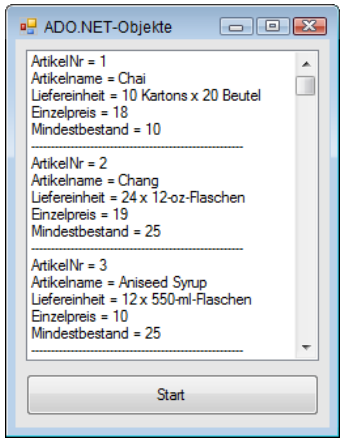
```
        Dim dt As DataTable = ds.Tables("ArtikelListe")
```

Nachdem je eine Zeilen- und Spaltenvariable definiert sind, sorgen zwei ineinander verschachtelte *For Each*-Schleifen für den Durchlauf der Auflistungen, wobei auf *DataRow*- und *DataColumn*-Objekte zugegriffen wird:

```
        For Each cRow As DataRow In dt.Rows
            For Each cCol As DataColumn In dt.Columns
                ListBox1.Items.Add(cCol.ColumnName & " = " & cRow(cCol.Ordinal).ToString())
            Next
            ListBox1.Items.Add("----------------------------------------------------")
        Next
        MyBase.OnLoad(e)
    End Sub
End Class
```

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.

13.8.2 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.

```
Imports System.Data.OleDb

Public Class Form1
    Private conn As New OleDbConnection(
                                "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Nordwind.mdb;")
    Private ds As New DataSet()
    Private cmd As New OleDbCommand()
```

Aktionsabfrage starten:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim da As 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:

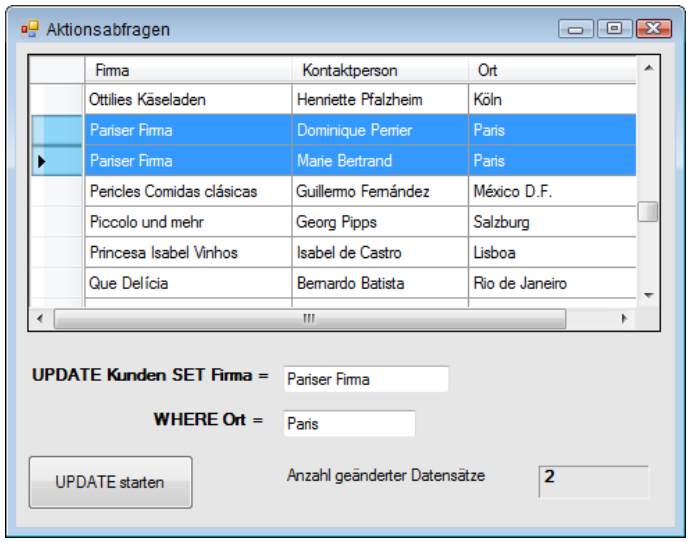
```
            Label1.Text = cmd.ExecuteNonQuery().ToString()
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
        da.Fill(ds, "Kunden")
        conn.Close()
```

Das *DataGridView* an das *DataSet* "anklemmen":

```
        DataGridView1.DataSource = ds
        DataGridView1.DataMember = "Kunden"
    End Sub
End Class
```

Test

Stimmt die Verbindungszeichenfolge des *Connection*-Objekts, dürfte es keine Probleme beim Ausprobieren unterschiedlicher Updates geben.



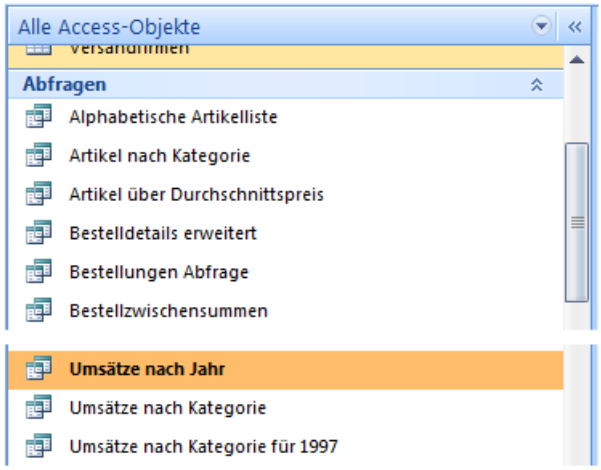
Bemerkungen

- 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 Praxisbeispiel

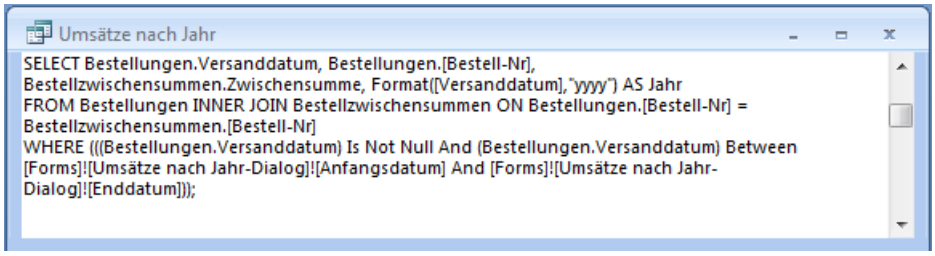
  ► 13.8.4 Die Datenbank aktualisieren

13.8.3 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

```
Imports System.Data.OleDb

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim connStr As String = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;"
        Dim conn As New OleDbConnection(connStr)

        Dim cmd As New OleDbCommand("[Umsätze nach Jahr]", conn)
        ' in [ ] einschließen!!!
        cmd.CommandType = CommandType.StoredProcedure
```

Die Definition der beiden Parameter und das Hinzufügen zur *Parameters*-Auflistung des *Command*-Objekts:

```
        Dim parm1 As New OleDbParameter("@Anfangsdatum", OleDbType.DBDate)
        parm1.Direction = ParameterDirection.Input
        parm1.Value = Convert.ToDateTime(TextBox1.Text)
        cmd.Parameters.Add(parm1)

        Dim parm2 As 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:

```
        Dim da As New OleDbDataAdapter(cmd)
        Dim ds As New DataSet()
        Try
            conn.Open()
            da.Fill(ds, "Jahresumsätze")
            conn.Close()
        Catch ex As Exception
            MessageBox.Show(ex.ToString())
        End Try
```

Die Anzeige:

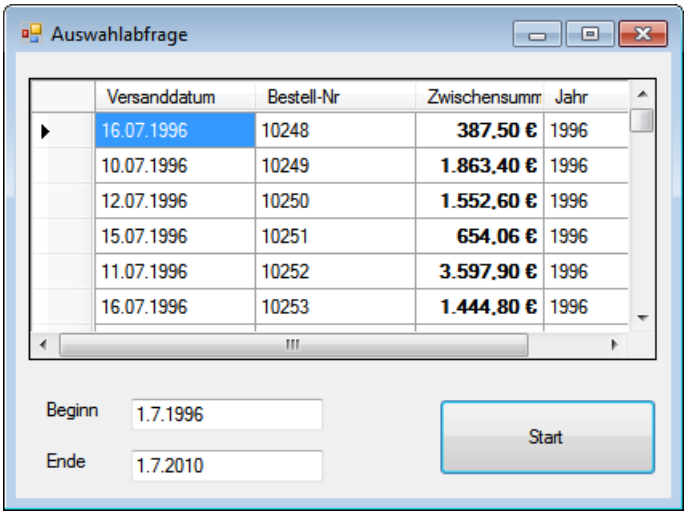
```
        DataGridView1.DataSource = ds
        DataGridView1.DataMember = "Jahresumsätze"
```

Wenigstens die Währungsspalte sollte eine ordentliche Formatierung erhalten (bei den übrigen Spalten belassen wir es bei den Standardeinstellungen):

```
        DataGridView1.Columns.Remove("Zwischensumme")
        Dim tbc As 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)
    End Sub
End Class
```

Test

Nach Eingabe sinnvoller Datumswerte dürfte sich Ihnen der folgende Anblick bieten:



13.8.4 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)

```
Imports System.Data.OleDb

Public Class Form1
```

Die wichtigsten Objekte sollten global verfügbar sein:

```
    Private conn As New OleDbConnection(
                              "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Nordwind.mdb;")
    Private da As OleDbDataAdapter = Nothing
    Private ds As DataSet = Nothing
```

Die folgende Methode *getArtikel* liefert ein gefülltes *DataSet* zurück:

```
    Public Function getArtikel() As DataSet
```

SelectCommand für *DataAdapter* erstellen:

```
        Dim selStr As String = "SELECT ArtikelNr, Artikelname, Einzelpreis, " &
                                        "Mindestbestand FROM Artikel ORDER BY Artikelname"
        da = New OleDbDataAdapter(selStr, conn)
```

Neu hinzugefügte Datensätze erhalten sofort einen Primärschlüssel:

```
        da.MissingSchemaAction = MissingSchemaAction.AddWithKey
        conn.Open()
        Dim ds As New DataSet()
        da.Fill(ds, "Artikel")
        conn.Close()
        Return ds
    End Function
```

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 Sub setArtikel(ByRef ds As DataSet)
        Dim cb As New OleDbCommandBuilder(da)                    ' !!!
```

```
        conn.Open()
        da.Update(ds, "Artikel")
        conn.Close()
    End Sub
```

Anzeigen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        DataGridView1.DataSource = Nothing
        ds = getArtikel()
```

DataGridView mit *DataSet* verbinden:

```
        DataGridView1.DataSource = ds
        DataGridView1.DataMember = "Artikel"
        formatDataGridView(DataGridView1)
    End Sub
```

Aktualisieren:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Nur die Änderungen zurück in die Datenbank schreiben:

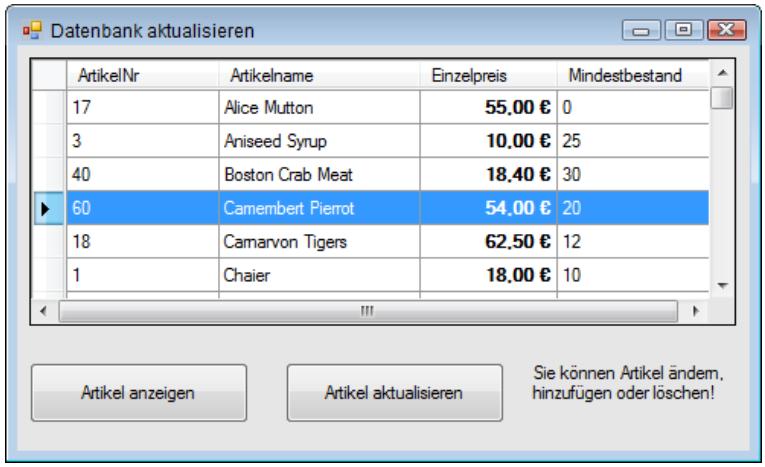
```
        Dim ds1 As DataSet = ds.GetChanges()
        If ds1 IsNot Nothing Then
            Try
                setArtikel(ds1)
```

Die per Referenz zurückgegebenen Datensätze werden mit dem Original-*DataSet* zusammengeführt:

```
                ds.Merge(ds1)
                ds.AcceptChanges()
                MessageBox.Show("Datenbank wurde aktualisiert!", "Erfolg")
            Catch ex As Exception
                ds.RejectChanges()
                MessageBox.Show(ex.Message, "Fehler")
            End Try
        End If
    End Sub
End Class
```

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.

13.8.5 Den ConnectionString speichern

Wenn Sie die Verbindungszeichenfolge nicht starr in den Quelltext eintragen, sondern in der Konfigurationsdatei (*App.config* bzw. *<Anwendungsname>.exe.config)* hinterlegen, kann der spätere Programmnutzer die *config*-Datei mit einem Text- oder XML-Editor bearbeiten, um die Verbindungszeichenfolge seinen Bedürfnissen anzupassen (das Programm braucht also nicht erneut kompiliert zu werden). Auch weitere Einstellungen, wie z.B. SQL-Abfragen, können auf diese Weise flexibel gehalten werden.

Das vorliegende Beispiel demonstriert dies anhand der Access-Datenbank *Nordwind*.

Oberfläche

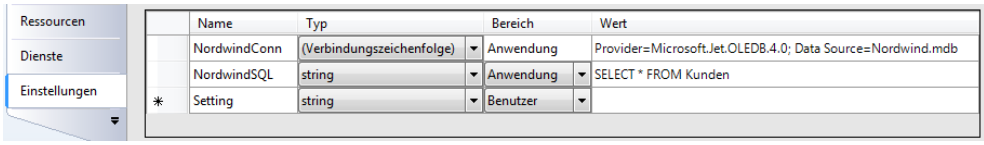
Öffnen Sie ein neues Projekt (Name hier *WindowsApplication1*). Auf dem Startformular finden ein *DataGridView* und ein *Button* ihren Platz.

Anwendungseinstellungen eintragen

Wir gehen für den Normalfall davon aus, dass sich die Access-Datenbank *Nordwind.mdb* im Anwendungsverzeichnis befindet. Außerdem soll standardmäßig die "Kunden"-Tabelle angezeigt werden.

Wählen Sie das Menü *Projekt/WindowsApplication1-Eigenschaften...* und öffnen Sie die Registerseite "Einstellungen".

Tragen Sie in die Liste die Einstellungen *NordwindConn* und *NordwindSQL* ein:



Für alle beiden Einträge wird als Bereich *Anwendung* eingestellt.

Der Blick in die XML-Datei *app.config* zeigt uns, dass die Verbindungszeichenfolge als *<Anwendungsname>.My.MySettings.**NordwindConn*** in der *connectionStrings*-Sektion gelandet ist:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>   ...
    <connectionStrings>
        <add name="WindowsApplication1.My.MySettings.NordwindConn"
             connectionString="Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb" />
    </connectionStrings>
```

Der SQL-String findet sich hingegen in der *applicationSettings*-Sektion:

```
    ...
    <applicationSettings>
        <WindowsApplication1.My.MySettings>
            <setting name="NordwindSQL" serializeAs="String">
                <value>SELECT * FROM Kunden</value>
            </setting>
        </WindowsApplication1.My.MySettings>
    </applicationSettings>
</configuration>
```

Quellcode

```
Imports System.Data.OleDb

Public Class Form1
```

Zwei globale Variablen (für Verbindungszeichenfolge und SQL-Abfrage) sollen die in den Anwendungseinstellungen gespeicherten Werte entgegennehmen:

```
    Private connStr As String
    Private sqlStr As String
```

Hinter der Schaltfläche "Mit Access-DB verbinden" liegt der Code zum Auslesen der entsprechenden Anwendungseinstellungen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

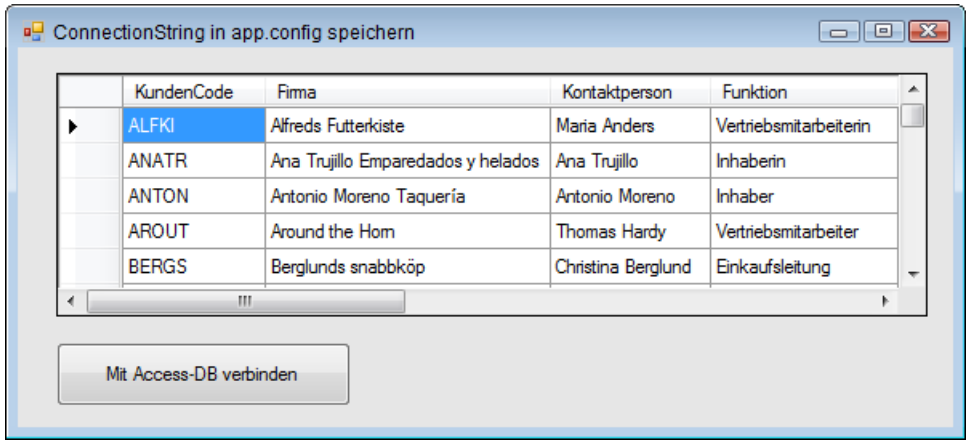
```
        connStr = My.MySettings.Default.NordwindConn
        sqlStr = My.MySettings.Default.NordwindSQL
        connect()
    End Sub
```

Die Verbindung zur Datenbank und die Anzeige im Datengitter:

```
    Private Sub connect()
        Dim conn As New OleDbConnection(connStr)
        Dim da As New OleDbDataAdapter(sqlStr, conn)
        Dim dt As New DataTable()
        Try
            conn.Open()
            da.Fill(dt)
            DataGridView1.DataSource = dt
        Catch ex As Exception
            DataGridView1.DataSource = Nothing
            MessageBox.Show(ex.Message, "Fehler")
        Finally
            conn.Close()
        End Try
    End Sub
End Class
```

Test

Nach dem Programmstart können Sie die Verbindung zur Datenbank herstellen.



Kapitel 14

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 33 (Windows Forms), 26 (WPF) ausgelagert.

14.1 Grundlegende Features des DataSets

Beim *DataSet* handelt sich um eine ziemlich 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.

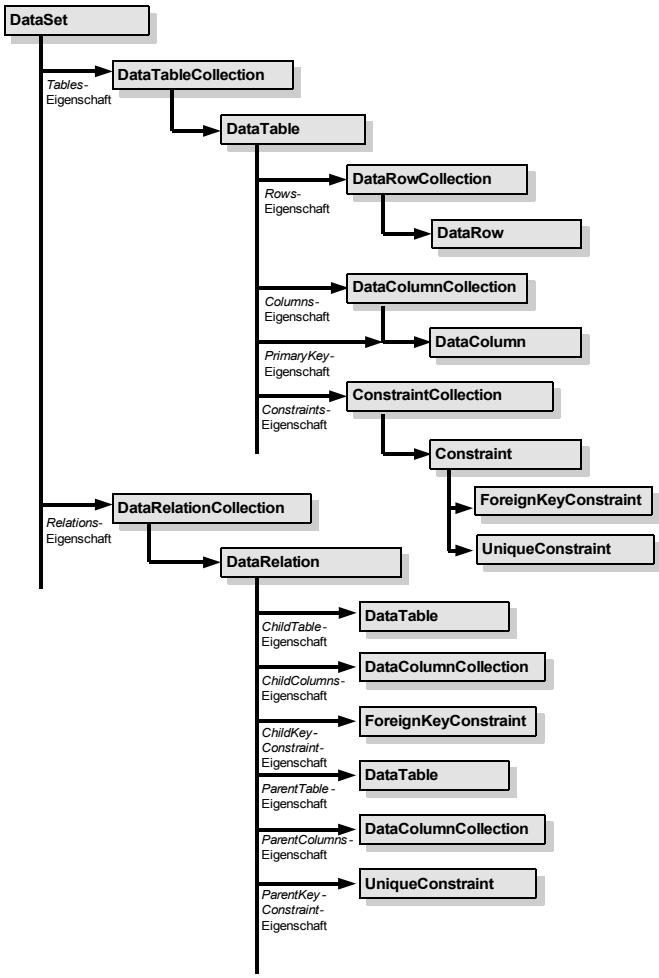
Es spielt auch keine Rolle, von welchem der .NET-Datenprovider das *DataSet* mit Daten gefüllt wurde oder ob es seine Daten auf andere Weise (z.B. aus einer XML-Datei oder direkt aus dem Programm) erhalten hat. Sie werden beim *DataSet*-Objekt auch keine Eigenschaften oder Methoden finden, die Aufschluss 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.

14.1.1 Die Objekthierarchie

Eine detaillierte Darstellung der Objekthierarchie zeigt die nachfolgende Abbildung.



14.1.2 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

14.1.3 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: `Dim myDataSet As New DataSet(Name As String)`

BEISPIEL 14.1: Das Erstellen einer *DataSet*-Instanz.

VB
```
Dim kuDS As 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 14.2: Erstellen einer exakten Kopie von *kuDS*.

VB
```
Dim copyDS As DataSet = kuDS.Copy()
```

Clone-Methode

Die mit der *Clone*-Methode erzeugte Kopie eines *DataSet*s enthält hingegen nur die Struktur- bzw. Schemainformationen.

BEISPIEL 14.3: Ein Klon eines *DataSet*s wird erstellt.

VB
```
Dim klonDS As DataSet = 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 14.4: Übergabe eines *DataRowState* beim Aufrufen von *GetChanges*.

VB
```
Dim changeDS As DataSet= custDS.GetChanges()                  ' alle Änderungen kopieren
Dim addedDS As DataSet = 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 14.5: Die Bedingung ist dann erfüllt, wenn zum DataSet *ds* neue Zeilen hinzugefügt wurden. Die neu hinzugefügten Zeilen werden in ein weiteres DataSet *ds1* kopiert.

VB
```
If ds.HasChanges(DataRowState.Added) Then
   Dim ds1 As DataSet = ds.GetChanges(DataRowState.Added)
End If
```

14.2 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.

14.2.1 DataTable erzeugen

Bereits im Vorgängerkapitel hatten wir im Zusammenhang mit der *Fill*-Methode des *DataAdapter*s häufig *DataTable*-Objekte erzeugt.

BEISPIEL 14.6: In ein vorhandenes *DataSet* wird mittels eines *DataAdapter*-Objekts *da* ein neues *DataTable*-Objekt *dt* mit der *TableName*-Eigenschaft "Artikel_Liste" eingefügt.

VB
```
da.Fill(ds, "Artikel_Liste")
```

Sie können eine neue *DataTable* auch direkt an die *Tables*-Collection des *DataSet*s anfügen.

BEISPIEL 14.7: Eine *DataTable* wird erzeugt.

VB
```
Dim dt As DataTable = ds.Tables.Add("Ku_Liste")
```

14.2.2 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 14.8: 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.

VB
```
Dim dt As New DataTable("Kunden")
Dim col As DataColumn = dt.Columns.Add("KundenCode", GetType(System.Int32))
col.AllowDBNull = False
col.Unique = True
dt.Columns.Add("Firma", GetType(System.String))
dt.Columns.Add("Kontaktperson", GetType(System.String))
dt.Columns.Add("Gehalt", GetType(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.

14.2.3 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: `Dim myRow As DataRow = myTable.NewRow()`

Anschließend lässt sich die neu hinzugefügte Zeile mit Hilfe von Spaltennamen oder Index bearbeiten.

BEISPIEL 14.9: Eine neue Zeile wird erzeugt, mit Werten gefüllt und zur *DataTable* hinzugefügt.

VB
```
Dim dt As DataTable = ds.Tables("KundenListe")
Dim rw As DataRow = dt.NewRow()
rw("Firma") = "Catering Service"                ' Zugriff über Spaltennamen
rw(2) = "Willy Schneider"                       '   ...   über Index
dt.Rows.Add(rw)                                 ' Hinzufügen zur DataTable
```

BEISPIEL 14.10: Sieht trickreich aus, aber es funktioniert

VB
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 14.11: *ImportRow*-Methode

VB
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

BEISPIEL 14.11: *ImportRow*-Methode

den Wert "Österreich" hat.

```
Dim kuGermDS As DataSet = kuDS.Clone()
Dim cRows() As DataRow = kuDS.Tables("Kunden").Select("Land = 'Österreich'")
Dim kuTable As DataTable = kuGermDS.Tables("Kunden")

For Each cRow As DataRow In cRows
     kuTable.ImportRow(cRow)
Next
```

14.2.4 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 14.12: Alle Zeilen und Spalten einer *DataTable* werden innerhalb von zwei geschachtelten *For Each*- Schleifen in eine *ListBox* ausgegeben.

VB
```
Dim dt As DataTable = ds.Tables("ArtikelListe")
For Each cRow As DataRow In dt.Rows
    For Each cCol As DataColumn In dt.Columns
           ListBox1.Items.Add(cCol.ColumnName & " = " & cRow(cCol.Ordinal))
    Next
    ListBox1.Items.Add("--------------------------------------------------")
Next
```

Den vollständigen Quellcode finden Sie im Praxisbeispiel

► 13.8.1 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 14.13: Die *Firma* des dritten Kunden eines *DataSet*s soll in einer *TextBox* ausgegeben werden.

VB
```
Dim dt As DataTable = ds.Tables("Kunden")
Dim rw As DataRow = dt.Rows(2)                    ' Auswahl der dritten 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 14.14: Der äquivalente Code zum Vorgängerbeispiel

VB
```
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 14.15: Der alternative Code zu den Vorgängerbeispielen, wobei aber die Zeilenauswahl über den Primärschlüssel erfolgt (die dritte Zeile hat den *KundenCode* ANTON).

VB
```
Dim dt As DataTable = ds.Tables("Kunden")
Dim colArr(1) As DataColumn                    ' Array mit einem Feld deklarieren
colArr(0) = dt.Columns("KundenCode")           ' Array füllen mit Primärschlüsselspalte
dt.PrimaryKey = colArr                         ' Primärschlüssel zuweisen
Dim rw As DataRow = 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 14.16: Die *Remove*-Methode der *Rows*-Collection erhält eine *DataRow* als Argument und entfernt diese aus der Auflistung.

VB
```
dt.Rows.Remove(rw)
```

BEISPIEL 14.17: Die *Delete*-Methode einer *DataRow* wird aufgerufen, um deren *RowState* in *Deleted* zu ändern.

VB
```
rw.Delete
```

14.3 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.

14.3.1 Erzeugen eines DataView

Dem üblichen Konstruktor wird das zugrunde liegende *DataTable*-Objekt übergeben:

SYNTAX: `Dim dv AS New DataView(dt As DataTable)`

BEISPIEL 14.18: Zwei verschiedene Sichten der *Artikel*-Tabelle von *Nordwind.mdb* werden erzeugt.

VB
```
Dim conn As New OleDbConnection(
                          "Provider=Microsoft.Jet.OLEDB.4.0;  Data Source=Nordwind.mdb;")
Dim cmd As New OleDbCommand("SELECT * FROM Artikel", conn)
Dim da As New OleDbDataAdapter(cmd)
Dim ds As New DataSet()
conn.Open()
da.Fill(ds, "Artikel_Liste")
conn.Close()
Dim dt As DataTable = ds.Tables(0)
Dim dv1 As New DataView(dt)
...
Dim dv2 As New DataView(dt)
```

14.3.2 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 14.19: Aus der Tabelle *Artikel* werden alle Artikel mit einem Preis unter 20 Euro herausgefiltert und nach dem Artikelnamen (mit "Z" beginnend) umsortiert.

VB
```
Dim dt As DataTable = ds.Tables("Artikel")
Dim dv As New DataView(dt)
dv.RowFilter = "Einzelpreis < 20"
dv.Sort = "Artikelname DESC"
```

Den kompletten Code enthält das Praxisbeispiel

► 14.6.1 Im DataView sortieren und filtern

14.3.3 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 14.20: In der *Kunden*-Tabelle der *Nordwind*-Datenbank wird der erste Kunde mit der *Kontaktperson* "Hanna Moos" gesucht und in einem *DataGridView* markiert.

VB
```
Dim dv As DataView =  ds.DefaultViewManager.CreateDataView(dt)
dv.Sort = "Kontaktperson"                          ' in dieser Tabellenspalte wird gesucht
Dim i As Integer = dv.Find("Hanna Moos")           ' Zeile wird gesucht
DataGridView1.CurrentCell = DataGridView1.Rows(i).Cells(0)      ' Zeile wird vorn markiert
If i = -1 Then  MessageBox.Show("Keinen Datensatz gefunden!")
```

Ein entsprechendes Beispiel finden Sie im Praxisbeispiel

► 14.6.2 Suche nach Datensätzen

14.4 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.

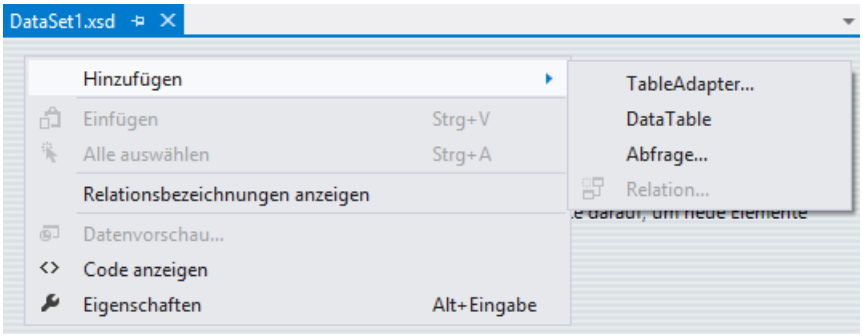
Zu einem typisierten DataSet gehören auch eine XML-Schema-Beschreibung (*.xsd*-Datei) sowie die Beschreibung der Anordnung der Daten in der grafischen Ansicht (*.xss*- und *.xcs*-Dateien).

14.4.1 Ein typisiertes DataSet erzeugen

Dank Assistenten-Unterstützung ist diese Arbeit in wenigen Schritten erledigt, die wir hier nur stichpunktartig aufzählen wollen.

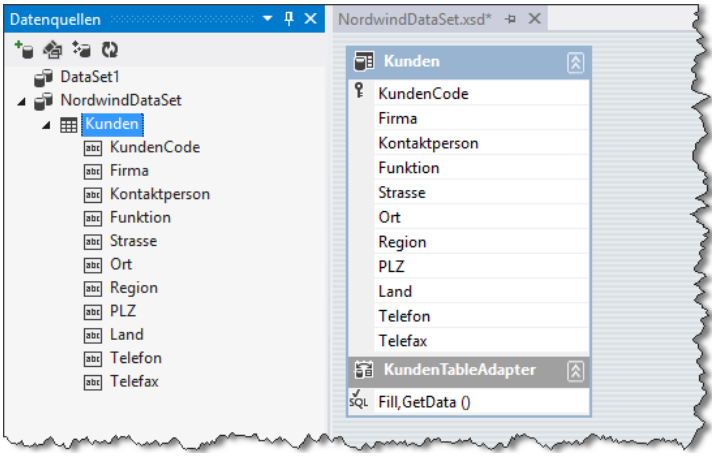
Wählen Sie das Menü *Projekt/Neues Element hinzufügen...* und die Vorlage "DataSet". Es wird eine *.xsd*-Datei (*DataSet1.xsd*) für ein typisiertes DataSet erzeugt.

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 14.21: Das typisierte DataSet *DataSet1* (siehe obige Abbildung) wird instanziiert, mit Daten gefüllt und angezeigt.

VB
```
Imports DataSet1TableAdapters                       ' Namespace für TableAdapter

Dim kta As New KundenTableAdapter()                 ' TableAdapter instanziieren
Dim ds1 As New DataSet1()                           ' typ. DataSet instanziieren
kta.Fill(ds1.Kunden)                                ' typ. DataSet füllen
DataGridView1.DataSource = ds1.Kunden               ' ... und anzeigen
```

14.4.2 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.

Eine schrittweise Anleitung zum Einrichten einer Datenquelle unter Windows Forms bietet Ihnen das Praxisbeispiel

► 35.8.1 Einrichten und verwenden einer Datenquelle

14.4.3 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 14.22: Eine Datenquelle, basierend auf der *Customers* Tabelle der *Northwind*-Datenbank, führt zum Generieren folgender typisierter Klassen:

VB
```
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 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 14.23: Vergleich der Schreibweisen beim Zugriff auf die *ContactName*-Spalte in der vierten Zeile der *Customers*-Tabelle der *Northwind*-Datenbank.

VB

Normales DataSet:

```
Dim cName As String = ds.Tables ("Customers").Rows(3)("ContactName").ToString
```

Typisiertes DataSet:

```
Dim cName As String = 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 14.24: Die (typisierte) Tabelle *northwindDataSet.Customers* wird mit Daten gefüllt.

VB
```
Me.CustomersTableAdapter.Fill(Me.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 14.25: Der Abfragemethode *FillByCity* wird als Parameter der Name einer Stadt aus einer *TextBox* übergeben.

VB
```
Me.CustomersTableAdapter.FillByCity (Me.NorthwindDataSet.Customers, Me.TextBox1.Text)
```

Da die Abfragemethoden streng typisiert sind, erhalten Sie beim Schreiben von Code die volle Unterstützung der IntelliSense von Visual Studio.

Eine Anleitung, wie Sie zu einem *TableAdapter* Abfragemethoden hinzufügen, erhalten Sie im Praxisbeispiel

► 35.8.1 Einrichten und verwenden einer Datenquelle

14.5 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

14.5.1 DataReader – der schnelle Lesezugriff

Am schnellsten geht das Lesen von Daten mit dem *DataReader* (siehe Kapitel 13), 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 13.3.1 oder Praxisbeispiel 13.8.2 "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.

14.5.2 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 14.4).

---

[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.

14.5.3 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[3].

Einen allerersten Eindruck von LINQ to SQL kann Ihnen das folgende Praxisbeispiel vermitteln:

► 14.6.5 Eine LINQ to SQL-Abfrage ausführen

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.

---

[1] Create, Retrieve, Update, Delete

[2] z.B. *Hibernate bzw. NHibernate*

[3] Leider konnten wir aus Umfangsgründen in diesem Buch nur auf die die allgemeinen Grundlagen von LINQ (siehe Kapitel 6) eingehen.

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 zurückzuschicken. Hier sollten Sie gezielt mit gekapselten *Command*-Anweisungen SQL-Befehle direkt gegen die Datenbank fahren.

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.

14.6 Praxisbeispiele

14.6.1 Im 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

```
Imports System.Data.OleDb

Public Class Form1
```

Das *DataView*-Objekt wrd global referenziert:

```
    Private dv As DataView = Nothing
```

Beim Laden der Anwendung erfolgt das Instanziieren und Initialisieren der übrigen Objekte:

```
    Protected Overrides Sub OnLoad(e As EventArgs)
        Dim conn As New OleDbConnection(
                       "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
        Dim cmd As New OleDbCommand(
              "SELECT ArtikelNr, Artikelname,Liefereinheit,Einzelpreis FROM Artikel", conn)
        Dim da As New OleDbDataAdapter(cmd)
        Dim dt As New DataTable()
```

DataTable füllen:

```
        da.Fill(dt)
```

Erzeugen des *DataView* in Standardansicht:

```
        dv = dt.DefaultView
```

Datengitter an *DataView* anbinden:

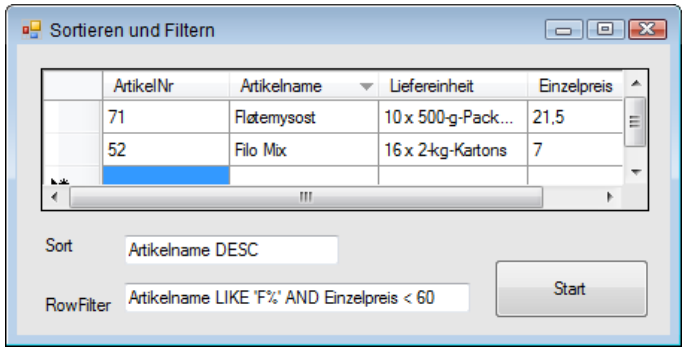
```
        DataGridView1.DataSource = dv
        MyBase.OnLoad(e)
    End Sub
```

Zum Filtern und Sortieren werden dem *DataView*-Objekt die Inhalte aus den beiden *TextBox*en zugewiesen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        dv.Sort = TextBox1.Text
        dv.RowFilter = TextBox2.Text
    End Sub
End Class
```

Test

Nach Programmstart zeigt die *DataView* zunächst alle Artikel in Standardansicht an. Nach dem Anklicken der "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.

14.6.2 Suche nach Datensätzen

Die im vorhergehenden Praxiosbeispiel 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

```
Imports System.Data.OleDb

Public Class Form1
    Private dv As DataView = Nothing

    Protected Overrides Sub OnLoad(e As EventArgs)
        Dim conn As New OleDbConnection(
                              "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=Nordwind.mdb;")
        Dim cmd As New OleDbCommand("SELECT * FROM Kunden", conn)
        Dim da As New OleDbDataAdapter(cmd)
        Dim dt As 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*:

```
        For Each c As DataColumn In dt.Columns
            ComboBox1.Items.Add(c.ColumnName)
        Next
```

Standardanzeige einstellen:

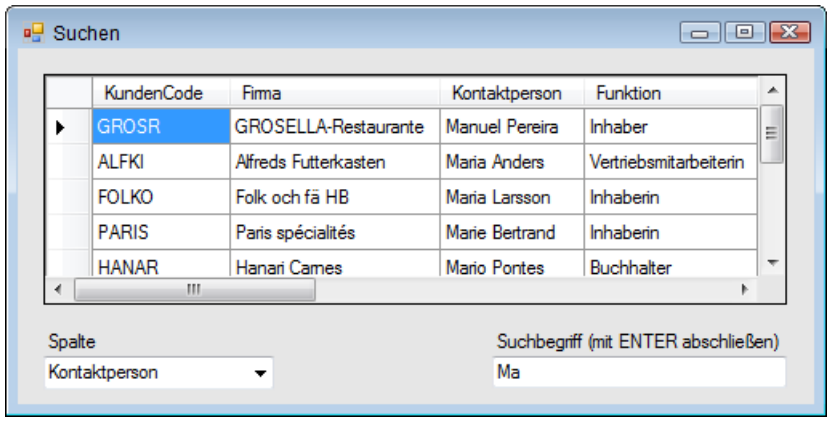
```
        ComboBox1.SelectedIndex = 1             ' Spalte "Firma"
        TextBox1.Text = "L"                     ' Default-Suchbegriff
        DataGridView1.DataSource = dv
        MyBase.OnLoad(e)
    End Sub
```

Die Suche startet nach Betätigen der *Enter*-Taste:

```
        Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter Then
            dv.Sort = ComboBox1.Text
            dv.RowFilter = dv.Sort & " LIKE '" & TextBox1.Text & "%'"
        End If
    End Sub
End Class
```

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.

14.6.3 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.

```
...
Imports System.Xml
Imports 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 Function ConvertDataSetToXML(ds As DataSet) As String

        Dim stream As MemoryStream = Nothing
        Dim writer As XmlTextWriter = Nothing
        Try
            stream = New MemoryStream()
```

XmlTextWriter mit dem *MemoryStream* initialisieren:

```
            writer = New XmlTextWriter(stream, System.Text.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):

```
           Dim arr() As Byte = stream.ToArray()
```

XML-String aus Byte-Array gewinnen und zurückgeben:

```
           Dim utf As New System.Text.UnicodeEncoding()
            Return utf.GetString(arr).Trim()
        Catch
            Return String.Empty
        Finally
            If writer IsNot Nothing Then writer.Close()
        End Try
    End Function
```

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 Function ConvertXMLToDataSet(xml As String) As DataSet
        Dim reader As StringReader = Nothing
        Try
            Dim ds As New DataSet()
            reader = New StringReader(xml)
```

XML-String in *DataSet* einlesen:

```
             ds.ReadXml(reader)
            Return ds
        Catch
            Return Nothing
        Finally
```

```
            If reader IsNot Nothing Then reader.Close()
        End Try
    End Function
```

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 zum Werkzeugkasten über Kontextmenü *Elemente auswählen ...*).

Quellcode Form1

```
...
Imports System.Xml
Imports System.IO

Public Class Form1
```

DataSet → XML-String:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim ds As DataSet = getTestDS()
        TextBox1.Text = ConvertDataSetToXML(ds)
        Button2.Enabled = True
    End Sub
```

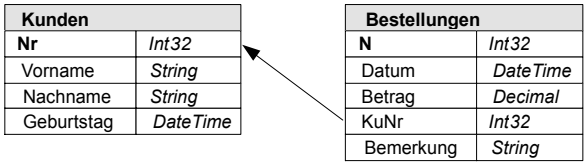
XML-String → DataSet:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim ds As DataSet = ConvertXMLToDataSet(TextBox1.Text)
        DataGrid1.DataSource = Nothing
        DataGrid1.DataSource = ds
    End Sub
```

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 Function getTestDS() As DataSet
```

Tabelle "Personen":

```
Dim dt1 As New DataTable("Personen")
```

Primärschlüssel:

```
Dim col1 As DataColumn = dt1.Columns.Add("Nr", GetType(System.Int32))
col1.AllowDBNull = False
col1.Unique = True
col1.AutoIncrement = True
col1.AutoIncrementStep = 1
```

Die restlichen Spalten hinzufügen:

```
dt1.Columns.Add("Vorname", .GetType(System.String))
dt1.Columns.Add("Nachname", GetType(System.String))
dt1.Columns.Add("Geburtstag", GetType(System.DateTime))
```

Zwei Datensätze hinzufügen:

```
Dim rw11 As DataRow = dt1.NewRow()
rw11("Vorname") = "Klaus"
rw11("Nachname") = "Müller"
rw11("Geburtstag") = Convert.ToDateTime("3.4.1975")
Dim rw12 As DataRow = 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":

```
Dim dt2 As New DataTable("Bestellungen")
Dim col2 As DataColumn = dt2.Columns.Add("Nr", GetType(System.Int32))
col2.AllowDBNull = False
col2.Unique = True
col2.AutoIncrement = True
col2.AutoIncrementStep = 1
dt2.Columns.Add("Datum", GetType(System.DateTime))
```

```
        dt2.Columns.Add("Betrag", GetType(System.Decimal))
        dt2.Columns.Add("PersNr", GetType(System.Int32))          ' Fremdschlüssel
        dt2.Columns.Add("Bemerkung", GetType(System.String))
```

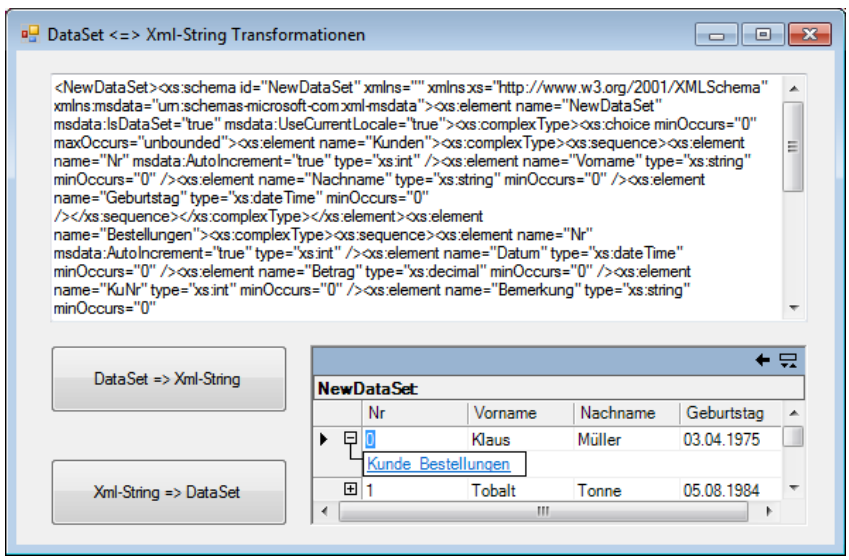
Zwei Datensätze hinzufügen:

```
        Dim rw21 As DataRow = dt2.NewRow()
        rw21("Datum") = Convert.ToDateTime("20.2.2006")
        rw21("Betrag") = Convert.ToDecimal("256,50")
        rw21("PersNr") = 0
        rw21("Bemerkung") = "per Nachname"
        dt2.Rows.Add(rw21)
        Dim rw22 As DataRow = dt2.NewRow()
        rw22("Datum") = Convert.ToDateTime("8.3.2006")
        rw22("Betrag") = Convert.ToDecimal("12,95")
        rw22("PersNr") = 0
        rw22("Bemerkung") = "per Scheck"
        dt2.Rows.Add(rw22)
```

DataSet zusammenbauen (mit 1 : n Relation zwischen *Kunden* und *Bestellungen*):

```
        Dim ds As New DataSet()
        ds.Tables.Add(dt1)
        ds.Tables.Add(dt2)
        ds.Relations.Add("Person_Bestellungen", ds.Tables("Personen").Columns("Nr"),
                                                ds.Tables("Bestellungen").Columns("PersNr"))
        Return ds
    End Function
End Class
```

Test



Zunächst lassen wir uns die XML-Darstellung des *DataSet*s in der *TextBox* anzeigen. Anschließend betätigen wir zwecks Rückkonvertierung die untere Schaltfläche.

14.6.4 Untypisierte in typisierte DataSets 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:

```
Dim ds As DataSet = CType(ds1, DataSet)
```

... funktioniert der folgende Code leider nicht:

```
Dim ds1 As DataSet1 = CType(ds, DataSet1)
```

Der "Dünnbrettbohrer" könnte allerdings mit folgendem Code sein Ziel erreichen:

```
ds.WriteXml("Temp.dat", XmlWriteMode.WriteSchema)
Dim ds1 As 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*.

```
...
Imports System.IO

Public Class Form1
    Public Function ConvertUntypedToTypedDS(ds As DataSet) As DataSet1
        Dim stream As MemoryStream = Nothing
        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 *MemoryStream*s einlesen:

```
            Dim ds1 As New DataSet1()
            ds1.ReadXml(stream, XmlReadMode.InferSchema)
            Return ds1
        Catch
            Return Nothing
        Finally
            If stream IsNot Nothing Then stream.Close()
        End Try
    End Function
```

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

► 14.6.3 Ein DataSet in einen XML-String serialisieren

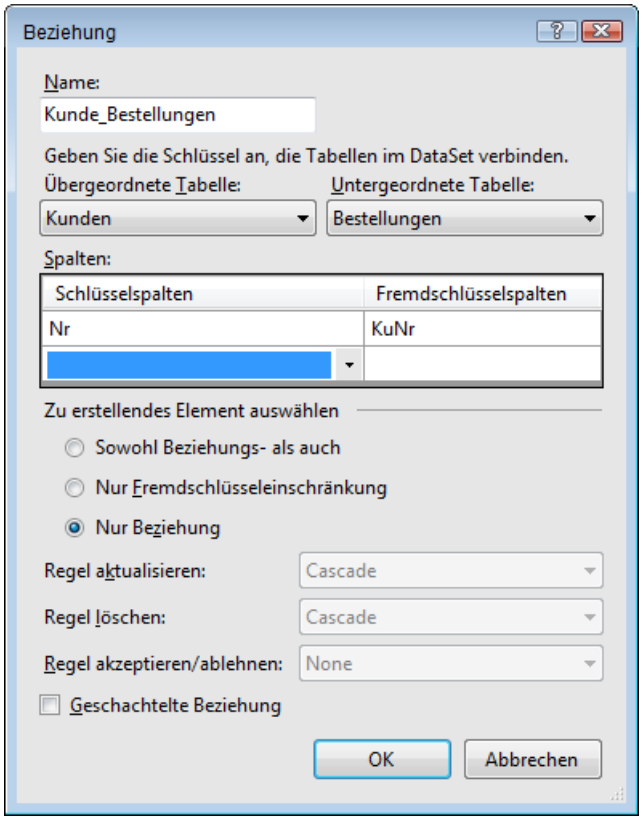
Typisiertes DataSet erzeugen

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.

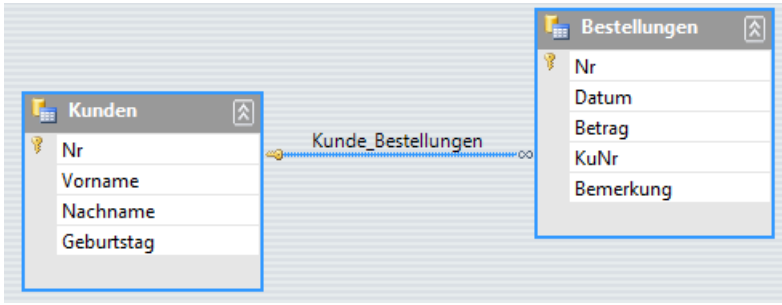
HINWEIS: Ein über das Projektmenü hinzugefügtes DataSet ist immer typisiert und automatisch eine Datenquelle!

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!).

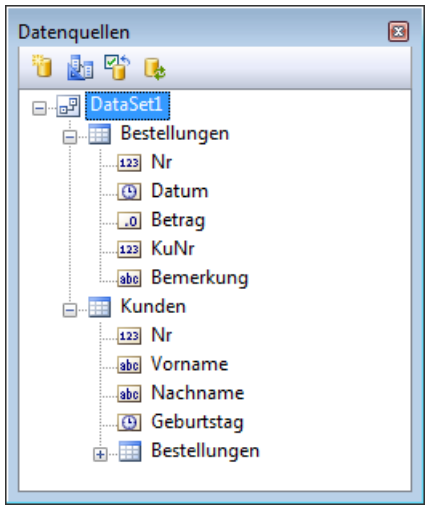


Nachdem auch die Tabelle *Bestellungen* fertig ist, wählen Sie im Kontextmenü *Hinzufügen/Relation...* und verbinden im Dialogfenster "Beziehung" beide Tabellen entsprechend der Abbildung.

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 *ConvertUntypedToTypedDS*:

```
...
Imports System.IO

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Das gefülltes untypisierte DataSet holen:

```
        Dim ds As DataSet = getTestDS()
```

Kontrollanzeige (umständlich weil nicht typisiert):

```
        MessageBox.Show(ds.Tables("Personen").Rows(0)("Vorname").ToString)
```

Typisiertes DataSet füllen:

```
        Dim ds1 As DataSet1 = ConvertUntypedToTypedDS(ds)
```

Datengitter mit dem typisierten DataSet verbinden:

```
        DataGrid1.DataSource = ds1
```

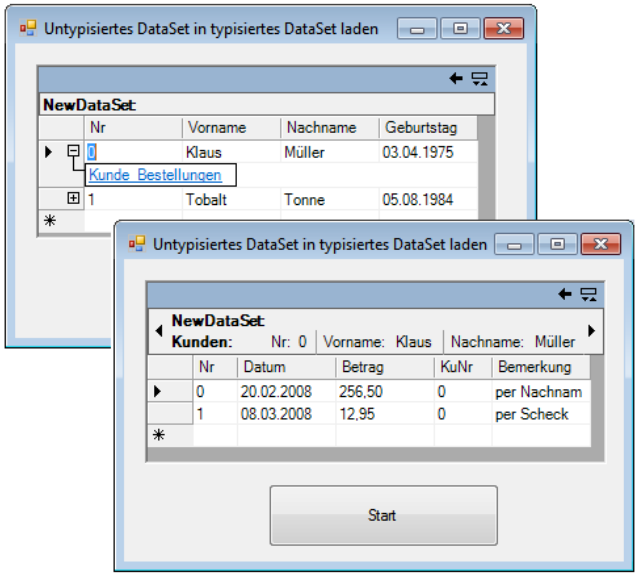
Kontrollanzeige (transparent weil typisiert):

```
        MessageBox.Show(ds1.Personen(0).Vorname)
    End Sub
    ...
End Class
```

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].

[1] Es kann einige Zeit dauern, bis der Dialog *Werkzeugkastenelemente auswählen* erscheint.

14.6.5 Eine LINQ to SQL-Abfrage ausführen

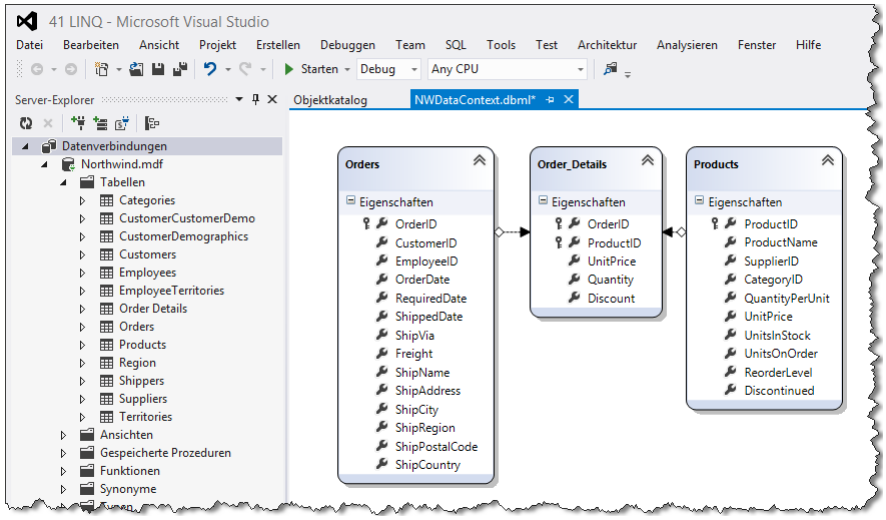
Mit LINQ to SQL steht Ihnen eine neue Technologie für den schnellen und einfachen Zugriff auf die Daten des Microsoft SQL Servers zur Verfügung.

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 obige Abbildung linke Seite).

HINWEIS: Für unser Beispiel fügen Sie die Tabellen *Order, Order_Detail* und *Product* ein.

Der Designer erstellt nachfolgend automatisch die erforderlichen VB-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

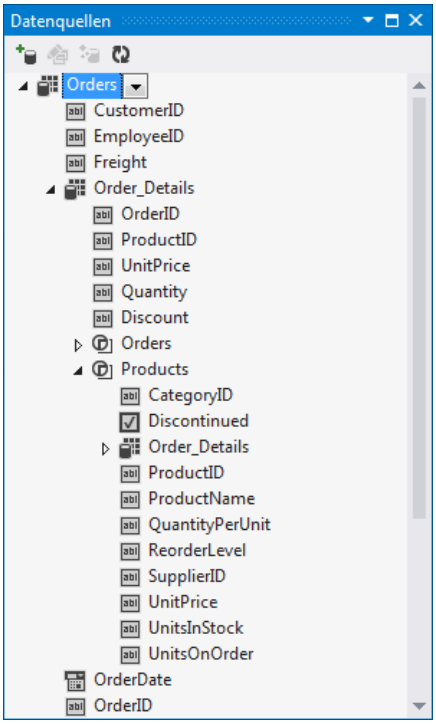
Möchten Sie Ihre Formulare mit Datenbindung ausstatten, von den Vorteilen eines Datenbanknavigators profitieren und schon zur Entwurfszeit die Darstellung von Tabellen etc. anpassen, kommen Sie nicht um die Definition einer Datenquelle herum[1].

Öffnen Sie das Fenster "Datenquellen" (Menü *Daten/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 wird dank Assoziation bereits unterhalb der Klasse *Orders* in der Liste der Datenquellen aufgeführt:

[1] Es sei denn, Sie wollen Unmengen von Code schreiben.



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 Class Form1
```

Instanz des *DataContext* erzeugen:

```
    Private dbnw As New NWDataContext()
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        OrdersBindingSource.DataSource = dbnw.Orders
```

Alternativ können Sie auch mit LINQ to SQL erst die Daten filtern/sortieren etc.:

```
        'Dim Orders = From o In dbnw.Orders ... Select o
        'OrdersBindingSource.DataSource = Orders
```

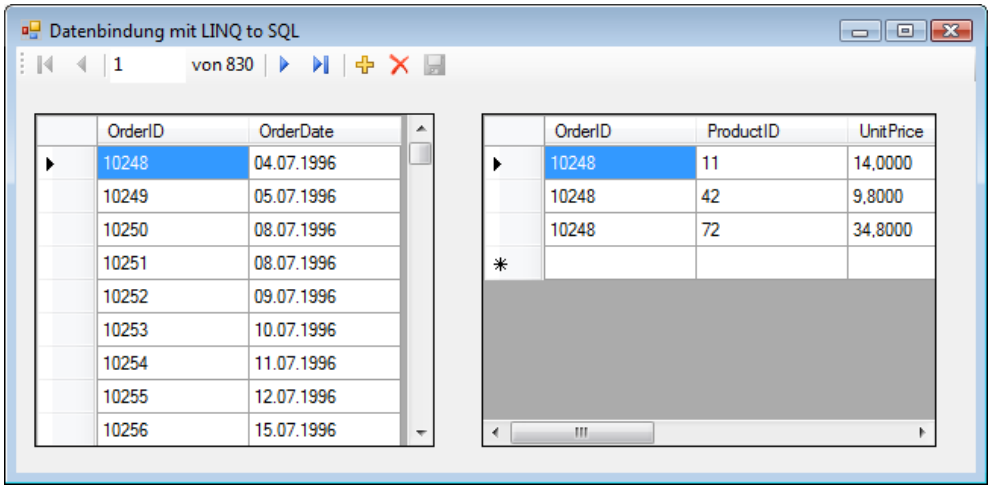
```
    End Sub
End Class
```

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 15

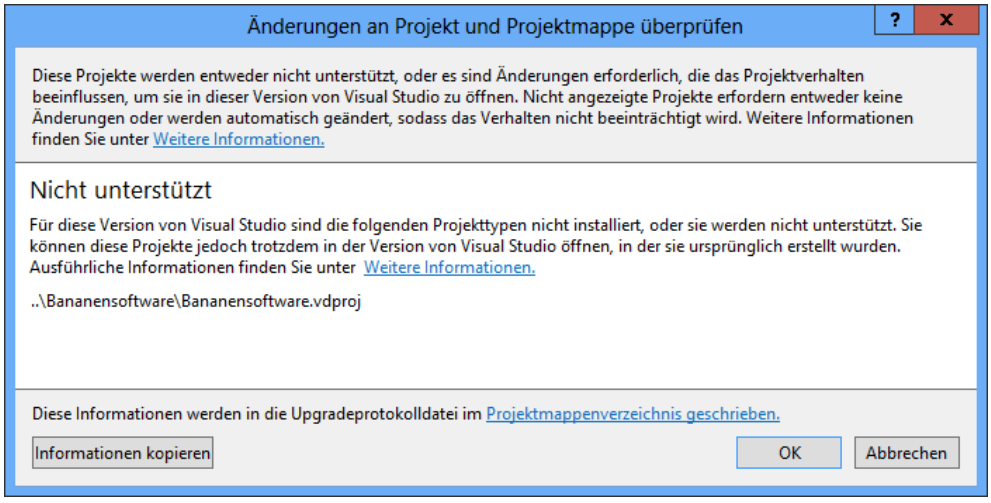
Verteilen von Anwendungen

Dem .NET-Programmierer bieten sich in Visual Studio 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 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 2012/2013/2015 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.

15.1 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.

15.1.1 Übersicht/Einschränkungen

Im Gegensatz zu den lokal installierten Windows-Anwendungen werden bei ClickOnce die Applikationen von einem zentralen Ort aus zur Verfügung gestellt. Dies kann eine Website, eine 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 dieser 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 auch diesmal mit einem einfachen Beispiel die Schritte zum fertigen ClickOnce-Projekt zeigen.

15.1.2 Die Vorgehensweise

Visual Studio bietet zumindest zwei Wege, um das *ClickOnce*-Deployment vorzubereiten:

- Manuelle Einstellung der Deployment-Optionen auf den Registerkarten *Sicherheit* und *Veröffentlichen*, die Teil der Projekteigenschaften sind, 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.

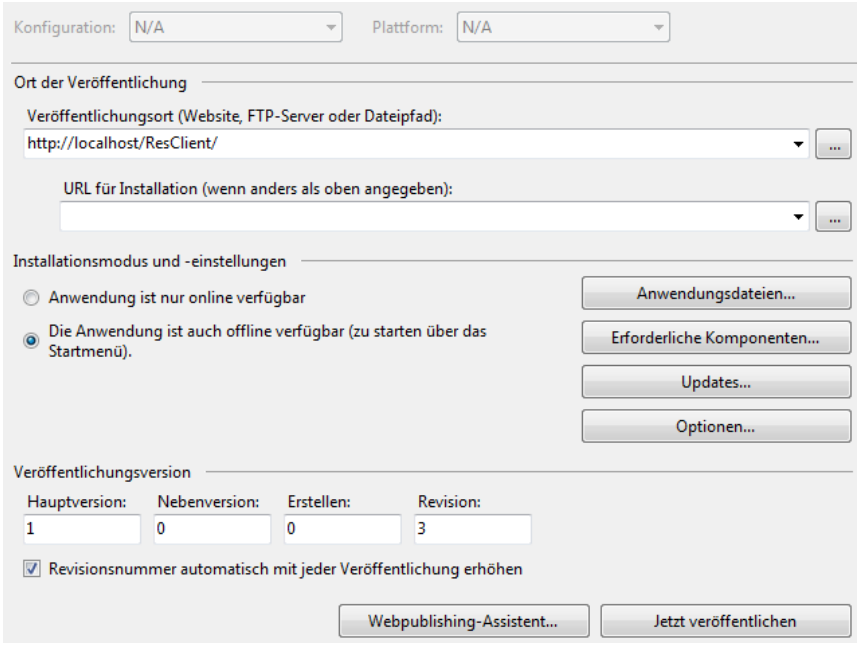
Im Folgenden wollen wir die Vorgehensweise am Beispiel einer aus zwei Komponenten bzw. Projekten (Client und Server) bestehenden Anwendung demonstrieren.

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.

15.1.3 Ort der Veröffentlichung

Rufen Sie das Menü *Projekt|ResClient*-Eigenschaften... des Projekts *ResClient* auf und wählen Sie die Registerseite *Veröffentlichen*:

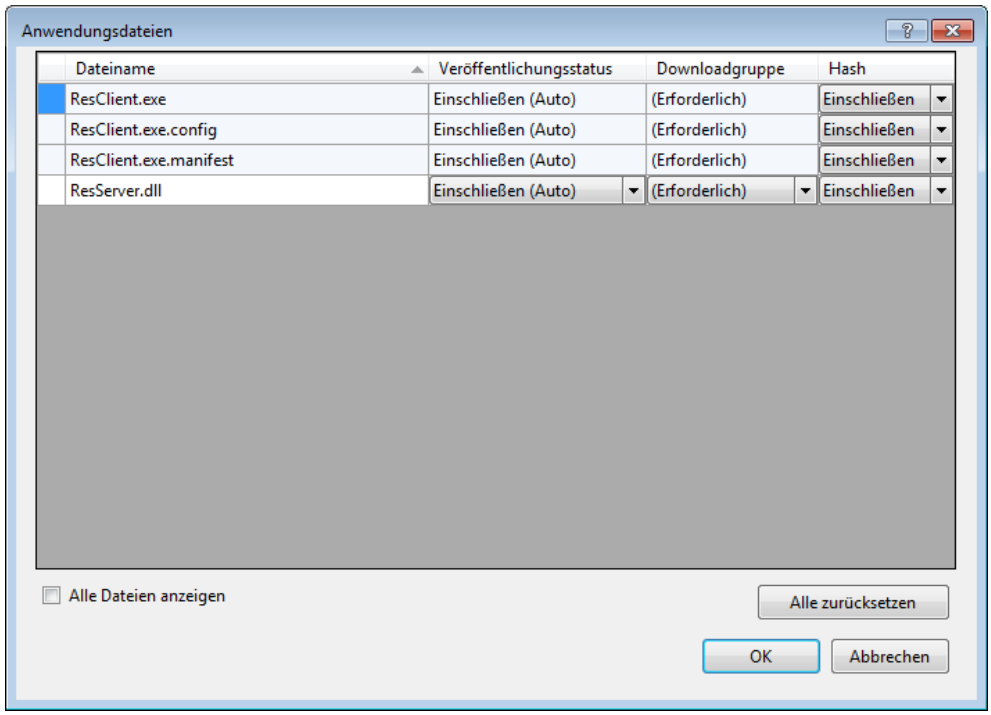


Tragen Sie dort den Ort der Veröffentlichung ein, für einen ersten Test eignet sich der lokale Webserver (IIS), alternativ können Sie auch einen Pfad oder einen FTP-Server angeben.

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.

15.1.4 Anwendungsdateien

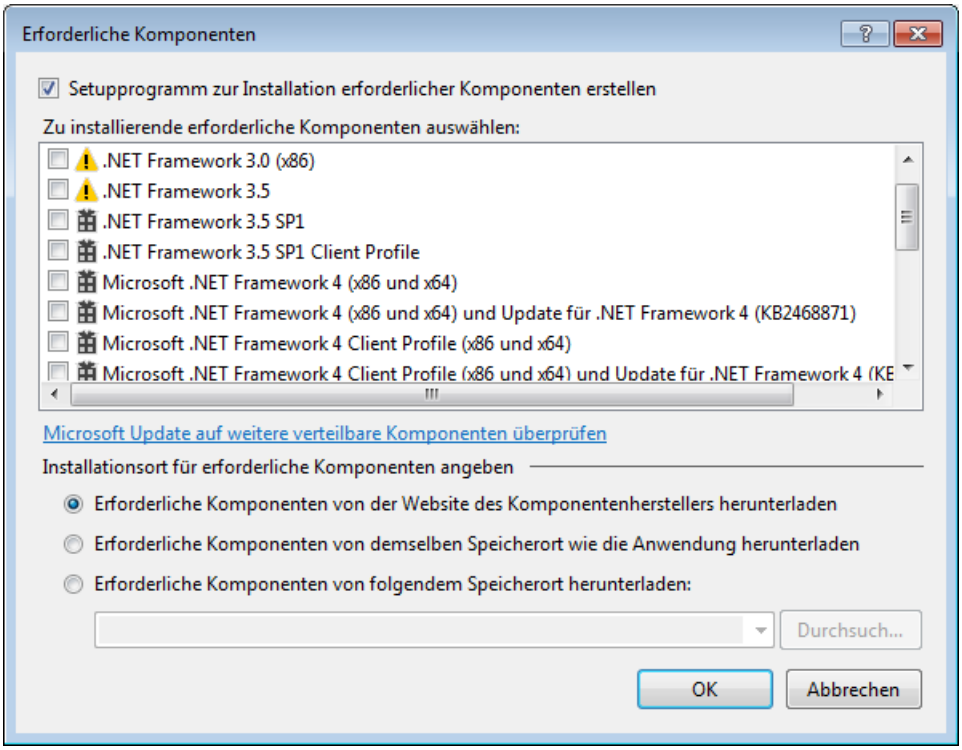
Klicken Sie auf die Schaltfläche *Anwendungsdateien...* so öffnet sich ein Dialog, in dem Sie angeben können, welche Dateien auf den Deployment-Server zu kopieren sind. *ClickOnce* erkennt selbstständig, welche Abhängigkeiten zwischen den Dateien bestehen und welche Dateien unbedingt erforderlich sind. Seien Sie also unbesorgt, wenn die für unser Beispiel erforderliche *ResServer.dll* nicht mit in der Liste aufgeführt ist.



15.1.5 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, 4.6* 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.



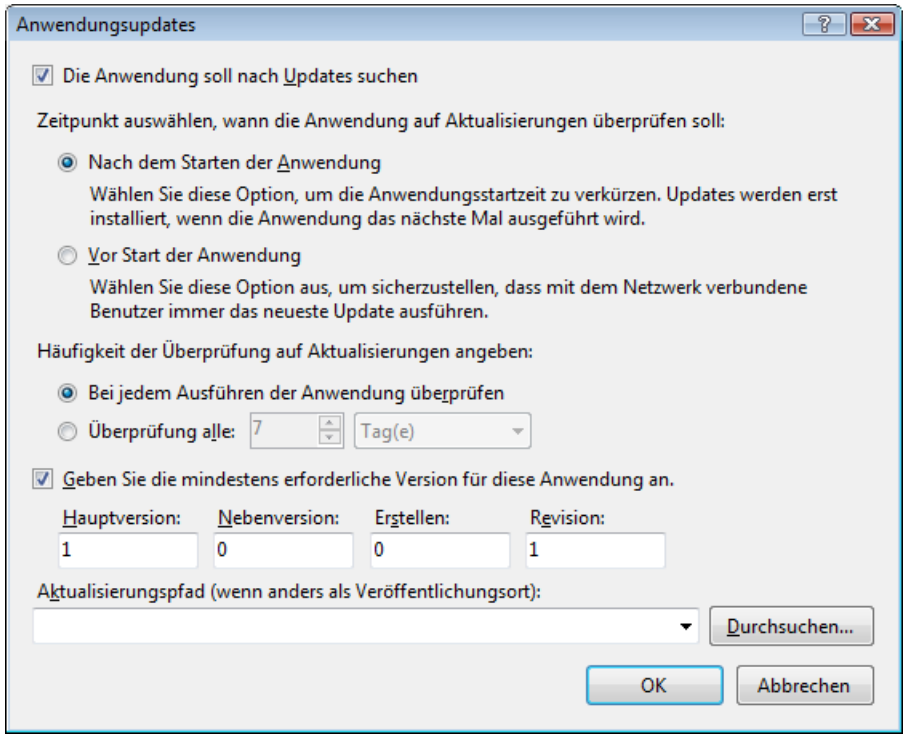
15.1.6 Aktualisierungen

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.

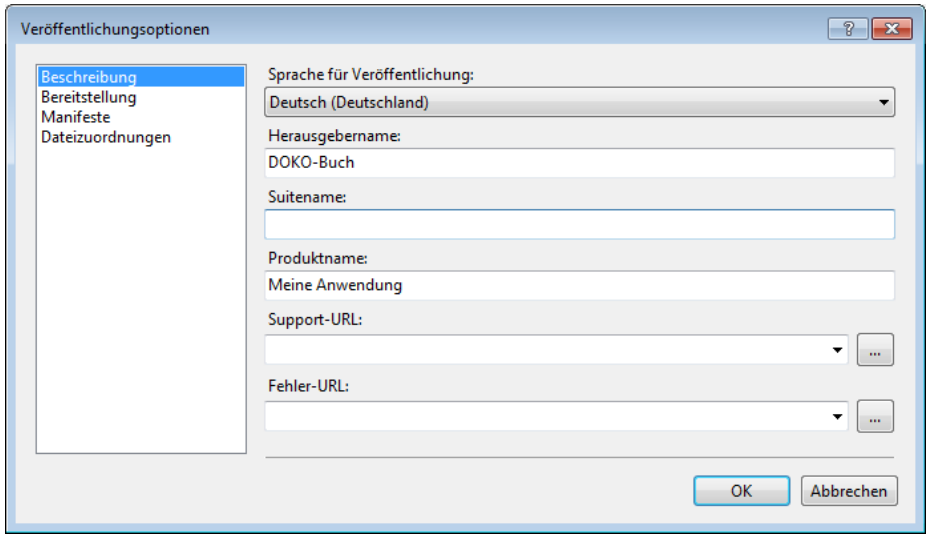
Sie entscheiden, ob (und wenn ja wann) auf Updates geprüft werden soll (vor oder nach dem Start der Anwendung). Bedenken Sie, dass Internet-Verbindungen nicht immer rasend schnell sein müssen, meist reicht es auch aus, nach dem Start zu prüfen und beim nächsten Programmstart steht die neue Version zur Verfügung.

Weiterhin können Sie das Update-Intervall, d h. die Häufigkeit der Updatesuche, bestimmen.



15.1.7 Veröffentlichungsoptionen

Legen Sie hier die wesentlichen Daten zu Ihrem Setup-Projekt fest (Herausgeber, Bereitstellungsseite, Dateizuordnungen etc.):

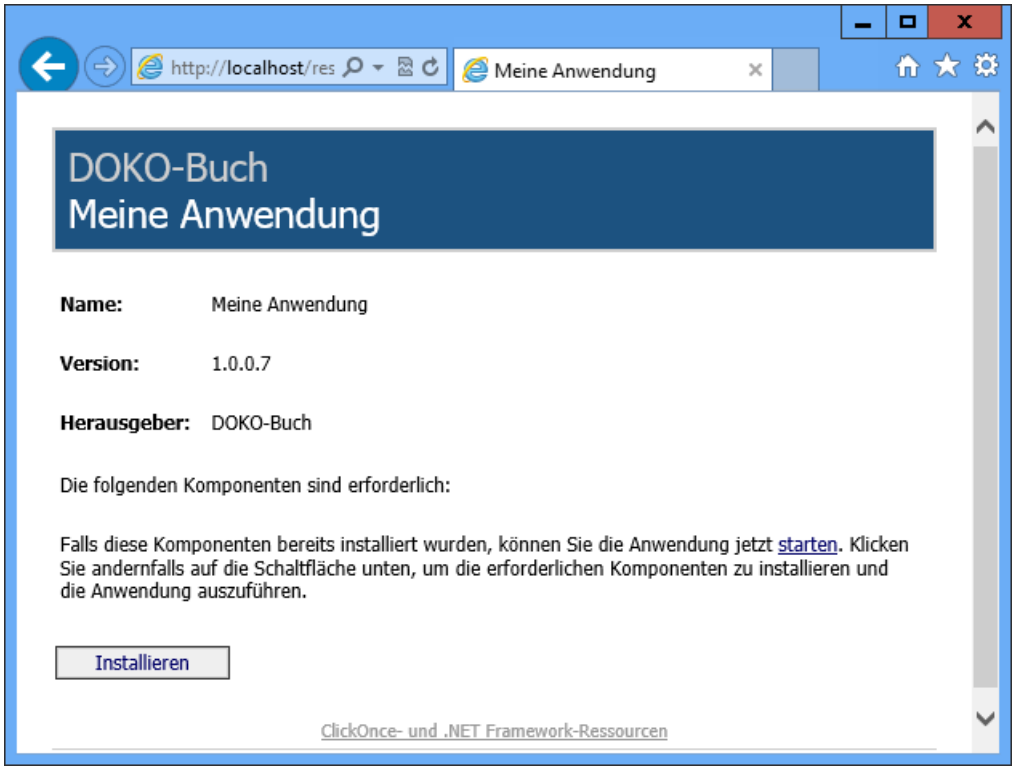


15.1.8 Veröffentlichen

Haben Sie alle Einstellungen vorgenommen, so steht dem entscheidenden Klick auf die Schaltfläche "Jetzt veröffentlichen" nichts mehr im Wege.

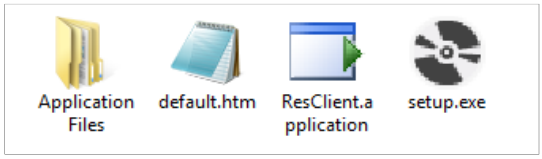
Es vergeht etwas Zeit, bis im Internet-Explorer die gewählte HTML-Seite (*default.htm*) erscheint, welche zusammen mit den Programmdateien auf dem Webserver liegt.

Die Installationswebsite:



15.1.9 Verzeichnisstruktur

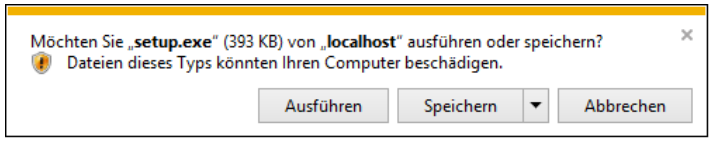
Auf dem Webserver wird das Verzeichnis *ResClient* mit der abgebildeten Struktur angelegt:

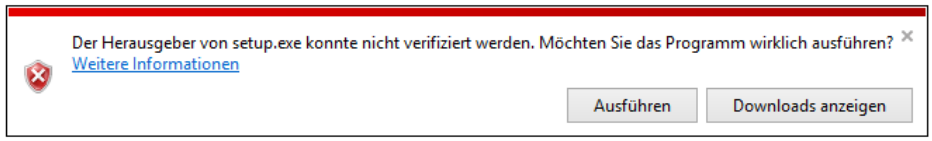


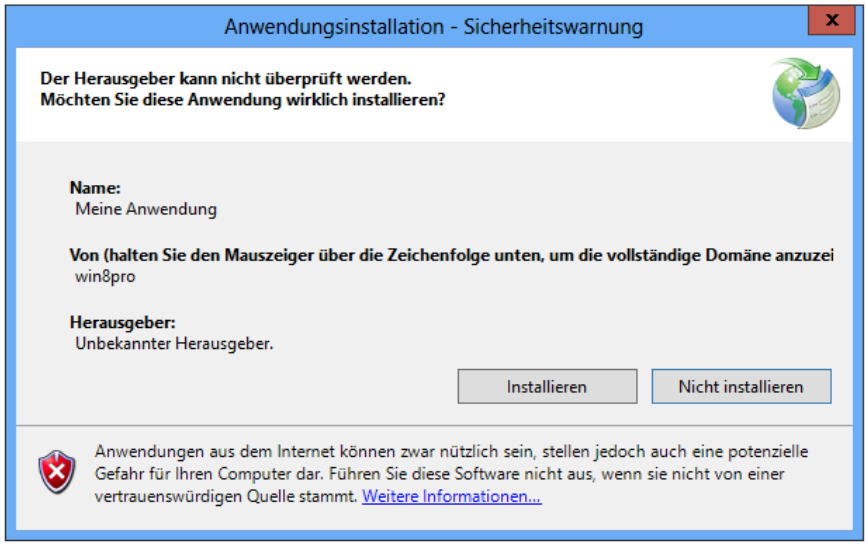
Ein paar Worte zu den einzelnen Dateien:

- 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*.

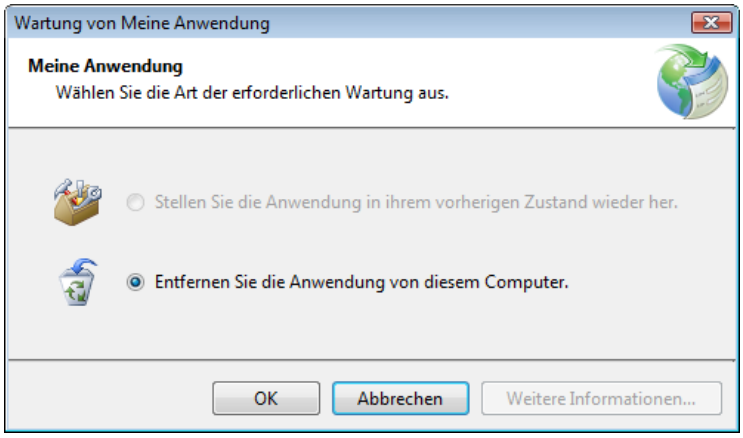
Es folgen mehrere Sicherheitswarnungen die Sie – zumindest für unser Beispiel – bedenkenlos ignorieren können (Schaltfläche *Ausführen* bzw. *Installieren* wählen):







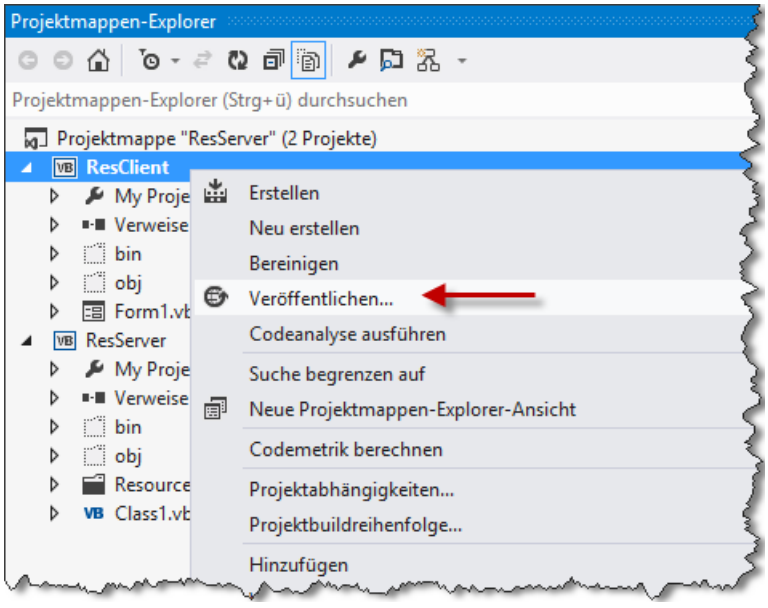
Nach dem Klick auf die *Installieren*-Schaltfläche und erfolgreichem Abschluss der Installation wird das Programm gestartet. Außerdem können Sie es – wie jedes andere Programm auch – über die *Start*-Schaltfläche des Windows Desktop aufrufen, selbst wenn keine Verbindung zum Netzwerk mehr besteht. Über *Systemeinstellungen|Software* lässt sich das Programm wieder deinstallieren, wobei Sie hier sogar die Option haben, die Vorgängerversion zu restaurieren.



15.1.10 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 *publish.htm*, und es geht weiter wie oben bereits beschrieben.

15.1.11 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.

15.2 InstallShield

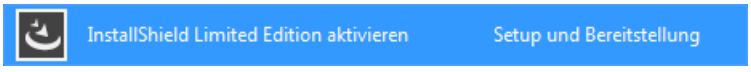
Nachdem wir uns im vorhergehenden Abschnitt intensiv mit dem ClickOnce-Deployment befasst haben, wollen wir uns nun mit dem von Visual Studio angebotenen InstallShield, dem Ersatz für die in Ungnade gefallenen klassischen Microsoft Setup-Projekte, beschäftigen. Mit Hilfe eines InstallShield-Projekts 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.

15.2.1 Installation

Möchten Sie ein neues Setup-Projekt erstellen (*Datei|Neu|Projekt|Vorlagen|Andere Projekttypen|Setup und Bereitstellung*), wird Ihnen in Visual Studio lediglich die Möglichkeit geboten, die Anwendung herunterzuladen und zu aktivieren.

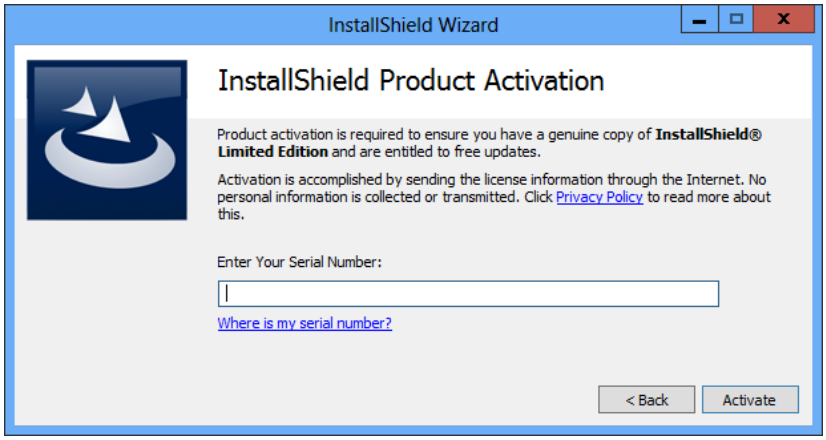


Klicken Sie auf obigen Eintrag, so landen Sie nach einigen Zwischenschritten auf der Flexera-Website, dem Anbieter von InstallShield.

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.

15.2.2 Aktivieren

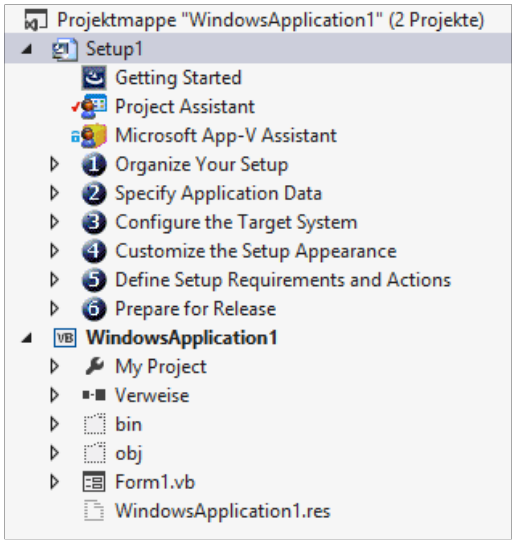
Rufen Sie jetzt den Menüpunkt *Datei|Neu|Projekt|Vorlagen|Andere Projekttypen|InstallShield Limited Edition* auf, können Sie InstallShield registrieren (den Key erhalten Sie per E-Mail):



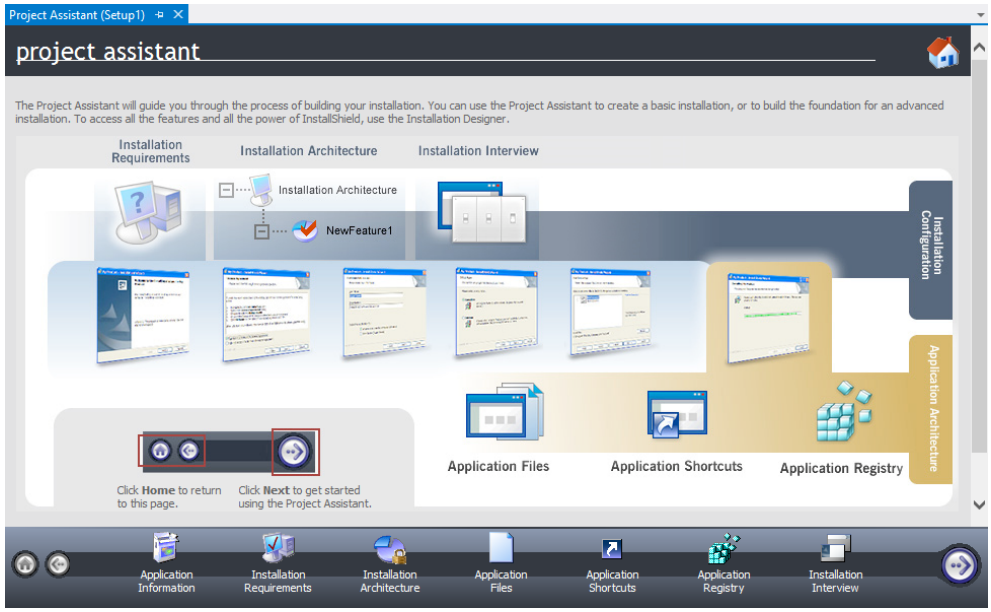
15.2.3 Ein neues Setup-Projekt

Öffnen Sie zunächst das VB-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 *InstallShield Limited Edition Project* (zur Projektmappe hinzufügen), wenn Sie eine Windows-Anwendung vertreiben möchten.

Nachfolgend finden Sie im Projektmappen-Explorer ein neues Projekt vor:

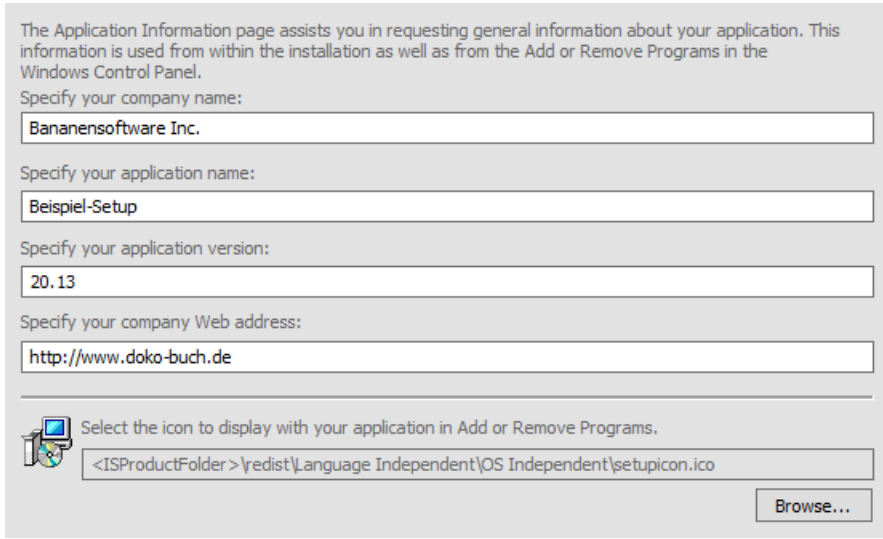


Gleichzeitig öffnet sich im Bearbeitungsbereich der Projekt-Assistent (ein recht buntes und wenig übersichtliches Fenster):



Klicken Sie auf die Pfeil-Schaltfläche in der rechten unteren Ecke und spezifizieren Sie die grundlegenden Informationen zu Ihrer Anwendung:

Anwendungseinstellungen



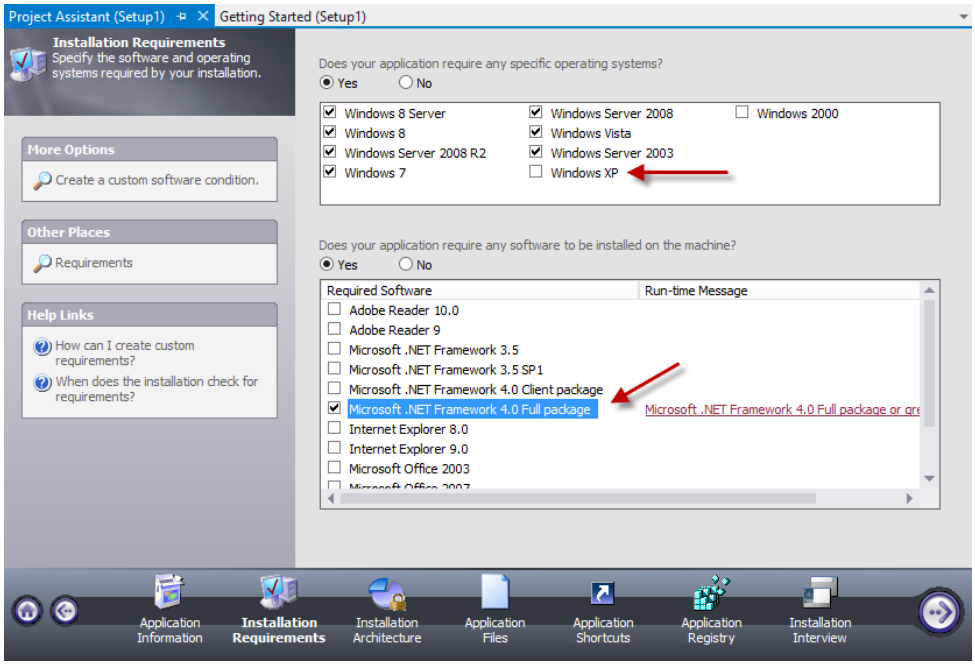
Die Bedeutung der einzelnen Eigenschaften dürfte recht schnell offenkundig sein.

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 einem solchen 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 Bedingungen in Ihr Setup-Programm einbauen, dazu dient ein eigener Assistent, der Sie dabei unterstützt,

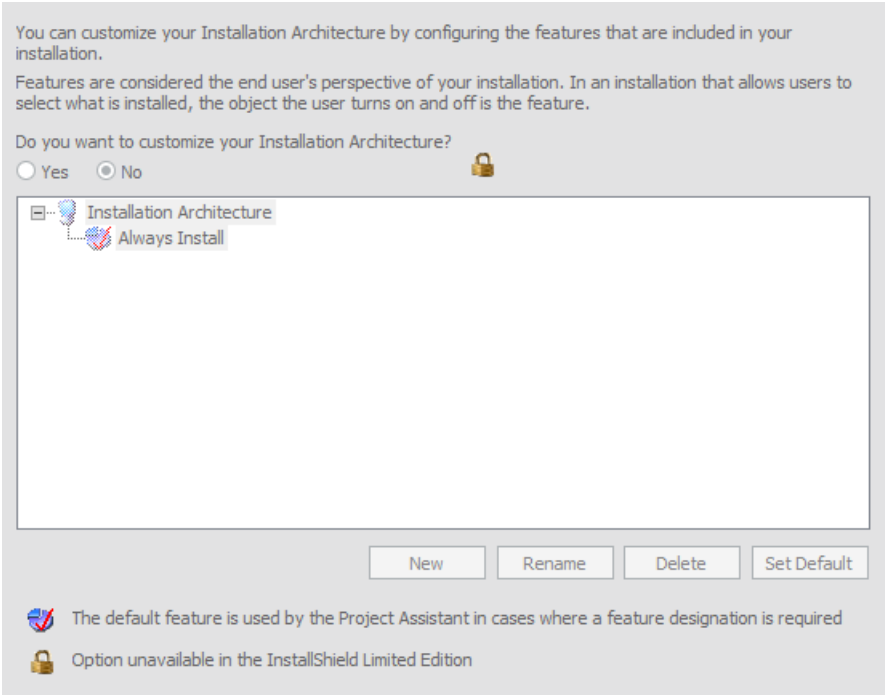
- Dateien,
- Verzeichnisse,

- 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:

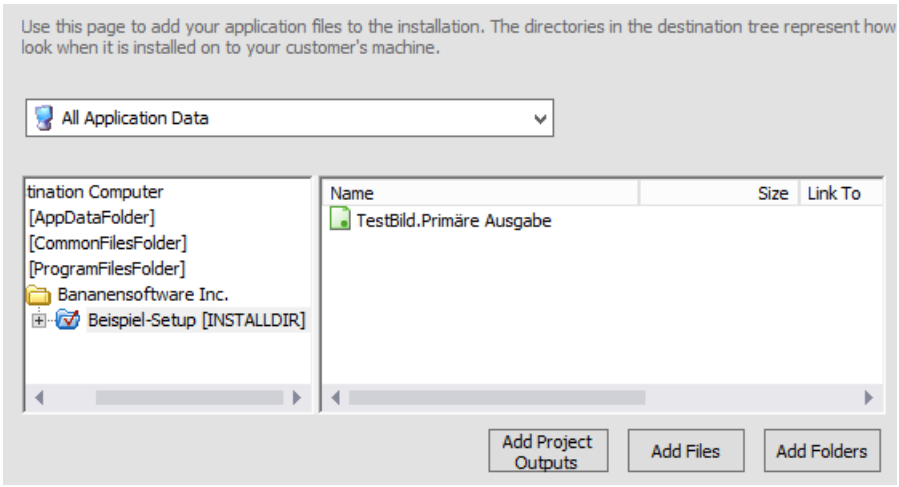


In diesem Fall können Sie verschiedene Optionsgruppen bilden, zwischen denen der Endanwender bei der Installation wählen kann (z.B. *Alles*, *Nur Anwendung* ...).

Anwendungsdateien

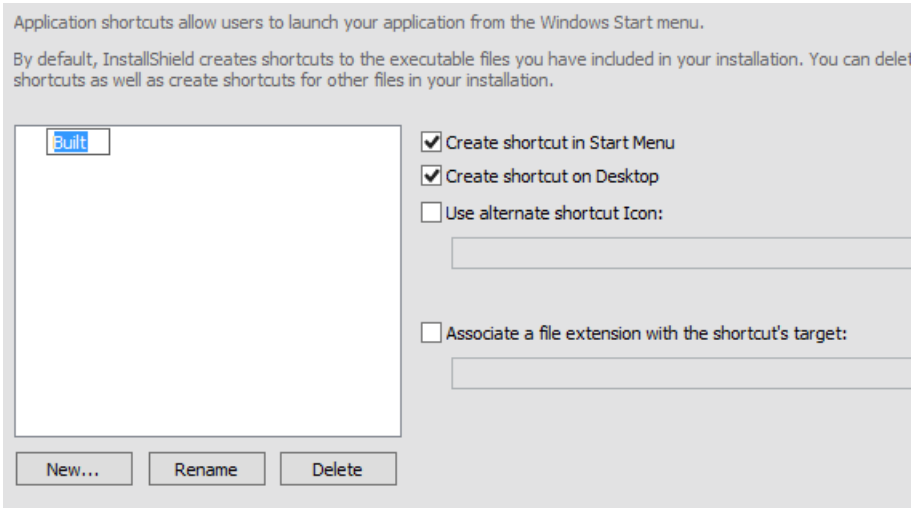
Wählen Sie im darauffolgenden Schritt, welche Dateien überhaupt auf dem Ziel-PC installiert werden sollen und in welchen Verzeichnissen dies zu erfolgen hat.

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].

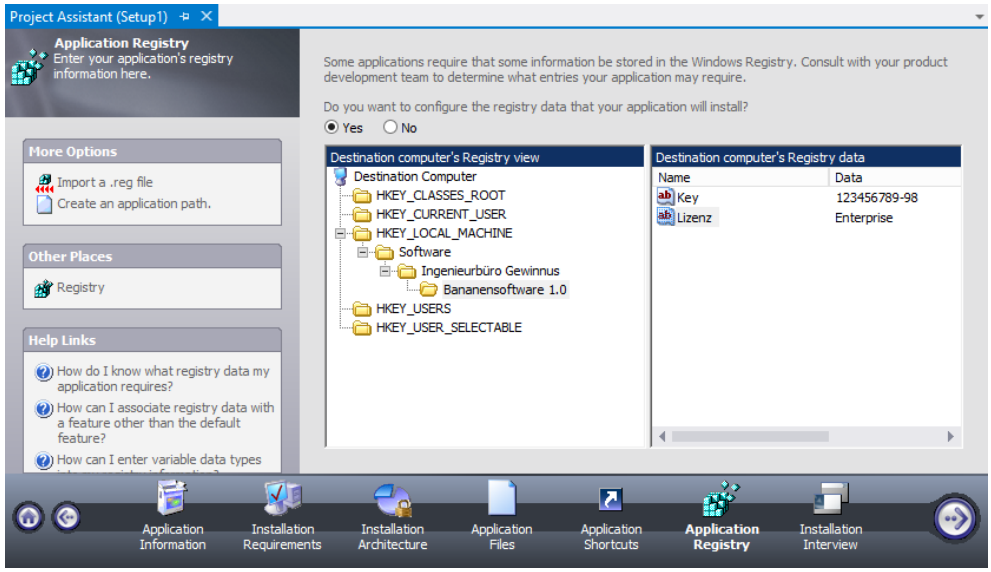


Optional können Sie hier auch noch ein anderes Symbol für die Verknüpfung festlegen.

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!

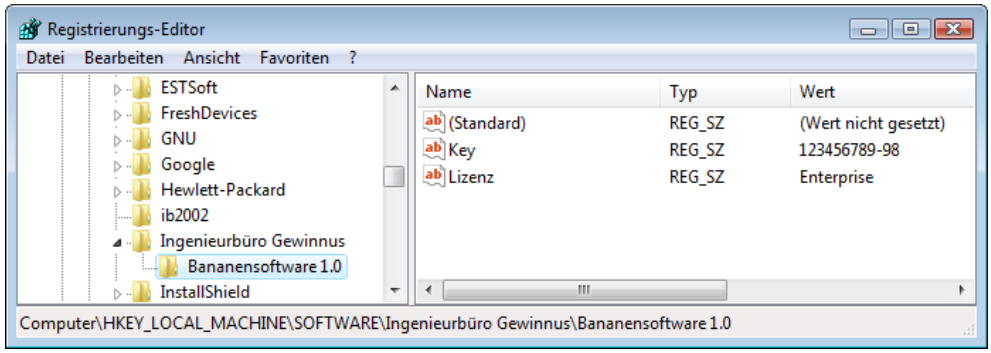


Legen Sie Einträge, die alle Nutzer betreffen (Installationspfade etc.), im Baumzweig HKEY_LOCAL_MACHINE ab, nutzerspezifische Daten (letzte geöffnete Datei etc.) im Baumzweig HKEY_CURRENT_USER. Für das Format der Registry-Einträge hat sich die folgende Notation eingebürgert:

```
…\<Software>\<Herstellername>\<Anwendungsname>\<Version>
```

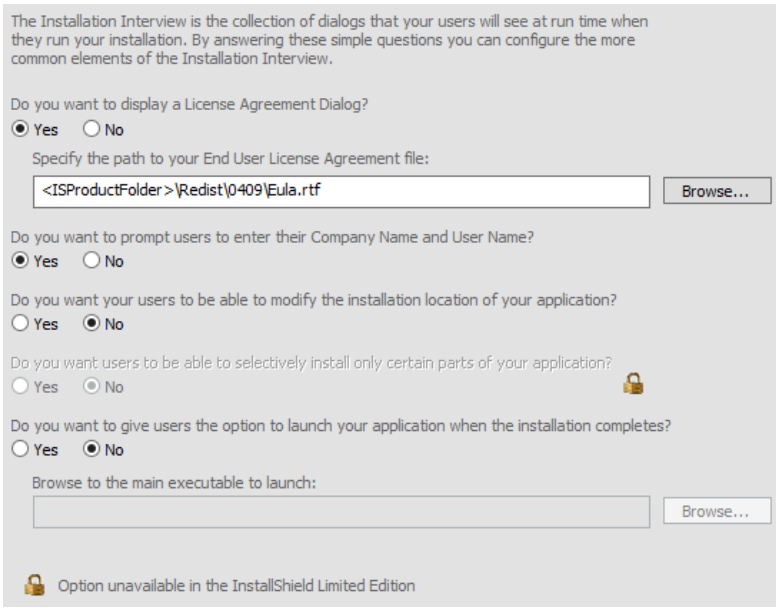
Im obigen Beispiel haben wir auch gleich noch zwei Values (*Key* und *Lizenz*) erzeugt.

Nach dem Generieren des Setup-Programms und der Installation finden sich die vordefinierten Registry-Einträge später auch an den gewünschten Stellen wieder:

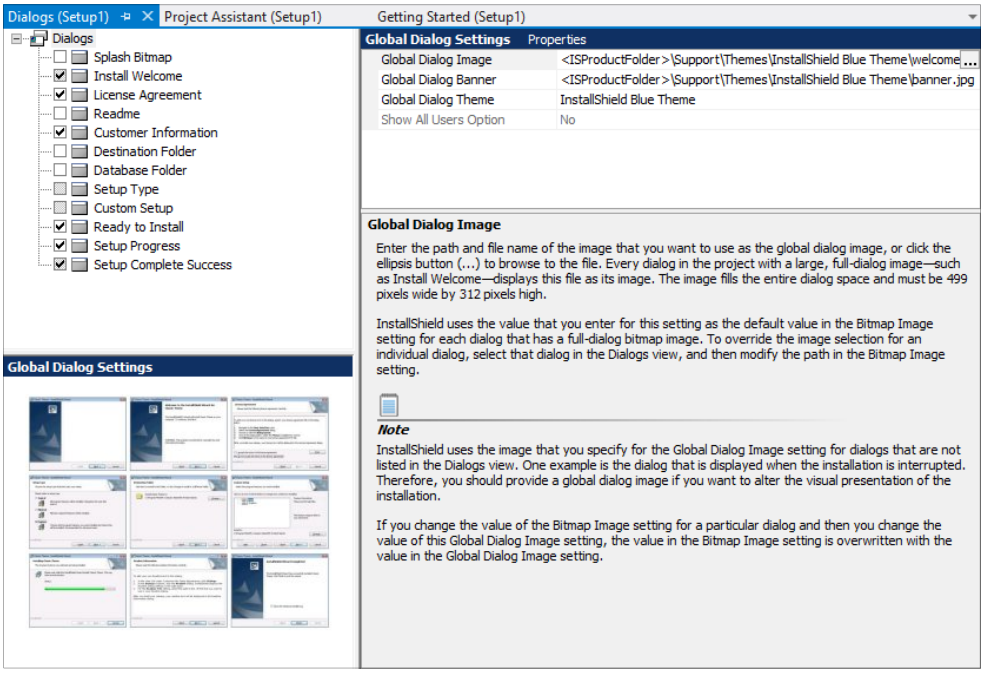


Setup-Dialoge/Erscheinungsbild

Welche Dialoge vom Setup benötigt werden, welche Lizenzbedingungen angezeigt werden etc., all dies lässt sich im letzten Schritt des Assistenten festlegen:



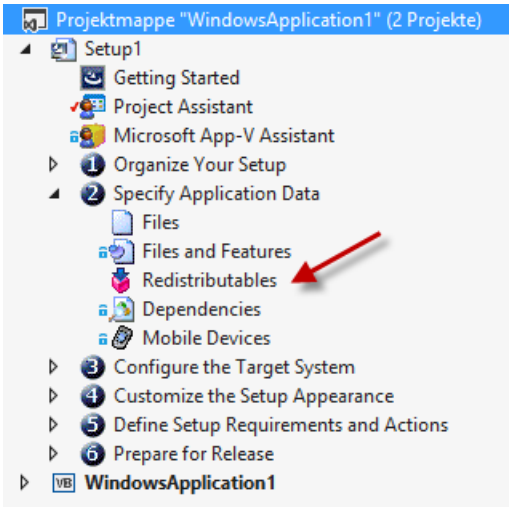
Von hier aus können Sie auch zur Dialogauswahl springen, eine Möglichkeit, die Ihnen auch per Projektmappen-Explorer offen steht.



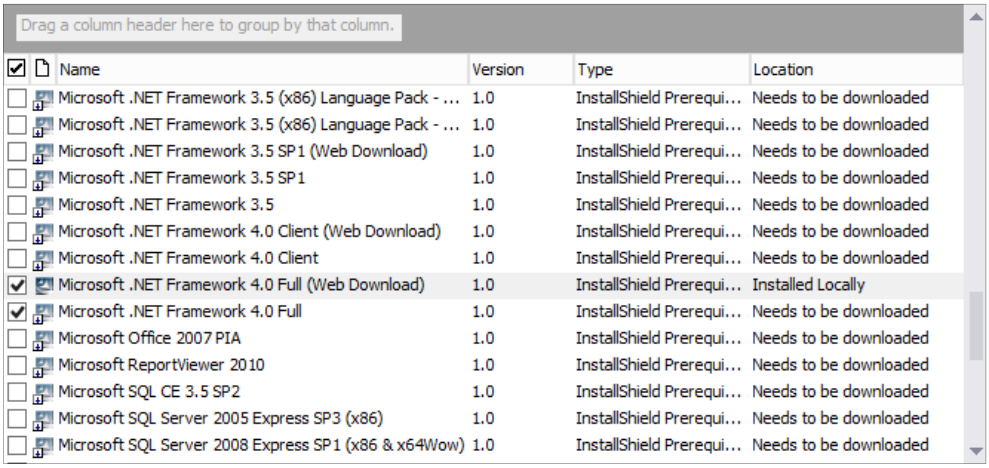
Installationsdateien

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]:

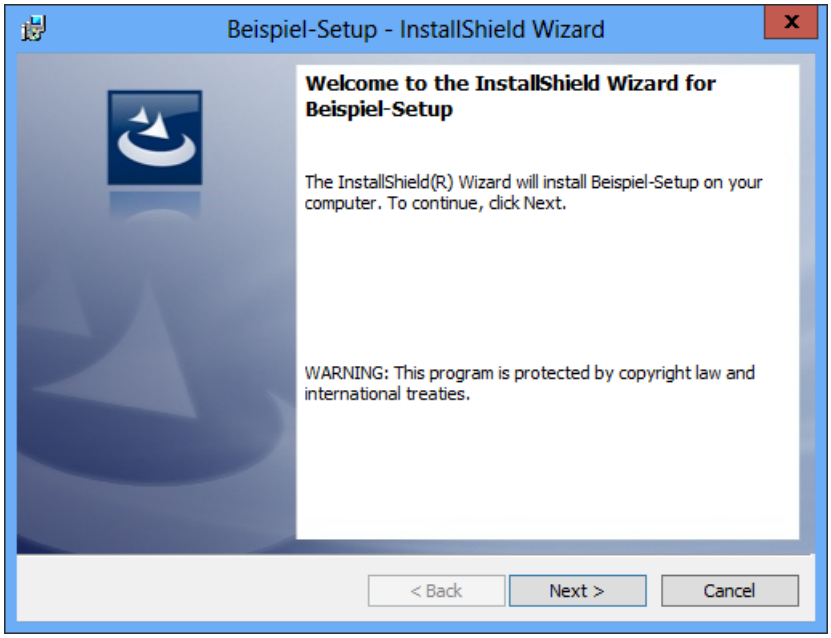


[1] In unserem Fall fehlte noch die Version 4 5, das aber dürfte sich bei Erscheinen des Buches geändert haben.

15.2.4 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.

15.3 Hilfedateien programmieren

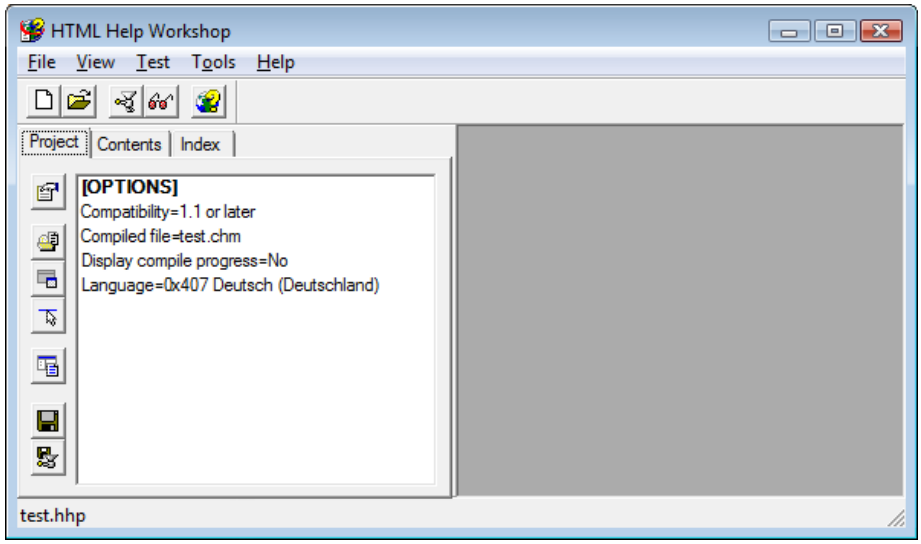
Zu einer professionellen VB-Applikation gehört, genauso wie zu jedem anderen Windows-Programm, eine Hilfedatei. In folgenden Abschnitt lernen Sie, wie man eine solche Hilfedatei unter Verwendung des Microsoft *HTML Help Workshops* erstellt.

Im Abschnitt 15.3.3 erfahren Sie dann, wie Sie die Hilfe in Ihr Programm integrieren.

15.3.1 Der HTML Help Workshop

Dieses komplexe Autorentool unterstützt Sie bei der Anfertigung von HTML-Hilfedateien:

- Erstellen eines Projekt-Files (*.hhp*)
- Formatieren von Hilfeseiten (*.htm, .html*)
- Erstellen von Inhaltsdateien (*.hhc*)
- Index-Files (Navigieren zwischen den Topics)
- Einbinden von Bild- und Multimediadateien
- Hilfefenster und Styles definieren
- Hilfedateien kompilieren (*.chm*)
- Testen und Debuggen von Hilfedateien



Den HTML Help Workshop können Sie unter folgender Adresse herunterladen:

LINK: `http://go.microsoft.com/fwlink/?linkid=14188`

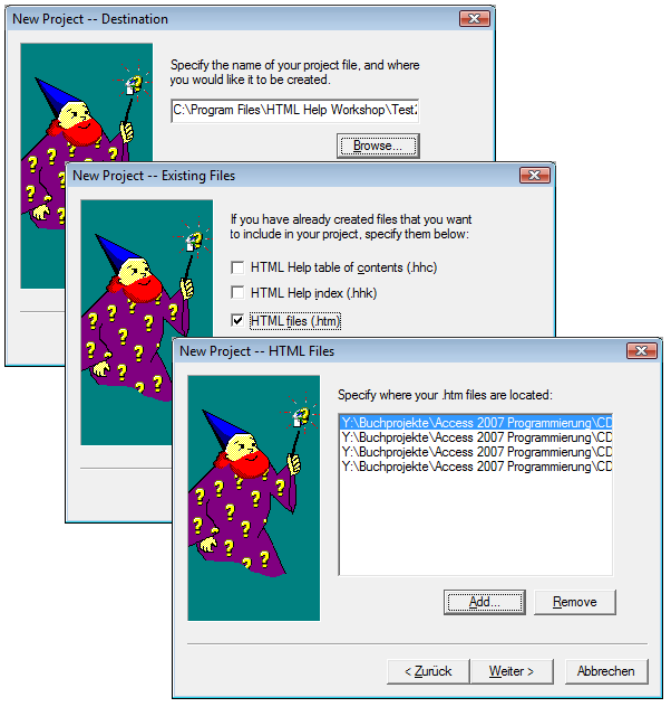
Die folgenden Ausführungen sollen Ihnen den Einstieg erleichtern.

HINWEIS: Auf das Erstellen von HTML-Seiten werden wir nicht weiter eingehen, dafür stehen zum Beispiel eine ganze Reihe von Free- bzw. Shareware-Tools zur Verfügung.

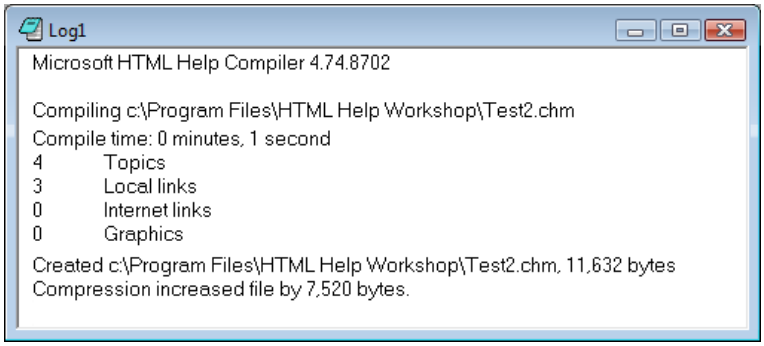
15.3.2 Bedienung am Beispiel

"Der Schuster trägt die schlechtesten Schuhe" und so ist auch die Hilfe zur HTML-Help teilweise ziemlich verwirrend. Hier der Versuch, einen besseren Einstieg zu vermitteln:

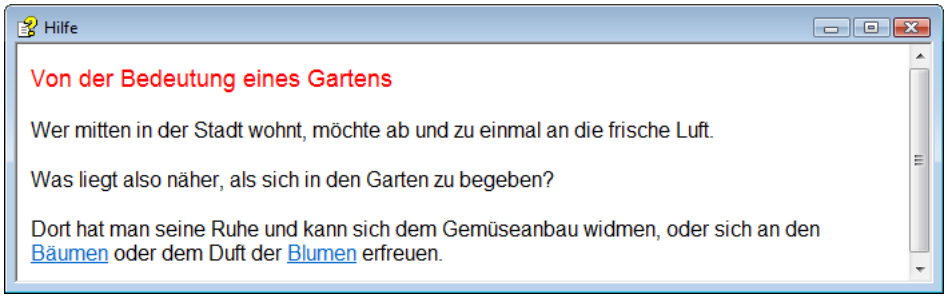
- Schreiben Sie mit einem HTML-Editor (z.B. Word oder *Microsoft FrontPage Express*) die einzelnen Hilfeseiten. Fügen Sie Hotspots bzw. Hyperlinks ein, so dass jede Seite erreichbar ist.
- Öffnen Sie den Microsoft HTML Workshop (*hhw.exe*).
- Wählen Sie im Dialogfeld *Project* den Befehl *File/New*.
- Unaufgefordert drängt Ihnen nun ein Wizard seine Dienste auf. An dem Dialogfeld mit der Option *Convert WinHelp Project* gehen Sie achtlos vorbei, das Häkchen setzen Sie nur in dem Fall, wenn Sie ein bereits existierendes älteres Hilfeprojekt (*.hpj*) in das neue HTML-Format konvertieren wollen. Dies dürfte besonders für den Umsteiger hilfreich sein, beantworten sich doch durch Vergleich des ursprünglichen mit dem konvertierten Projekt-File viele Fragen von selbst.
- Über die Schaltfläche *Browse* spezifizieren Sie Ihr *.hhp*-Projekt-File. Zweckmäßigerweise legen Sie es im gleichen Verzeichnis an, in welchem sich auch die *.htm*-Dateien befinden (z.B. als *Garten.hhp*).
- Wählen Sie die Option *Htm-Files*, da Sie die HTML-Dateien ja bereits erstellt haben.
- Im nächsten Fenster fügen Sie über *Add* die *.htm*-Dateien hinzu.



- Klicken Sie nacheinander auf die vorletzte ("Save project, contents and index files") und auf die letzte Schaltfläche ("Save all project files and compile") der linken (senkrechten) Symbolleiste des HTML Help Workshops. Im Logfenster (rechts) können Sie sich vom Erfolg überzeugen:

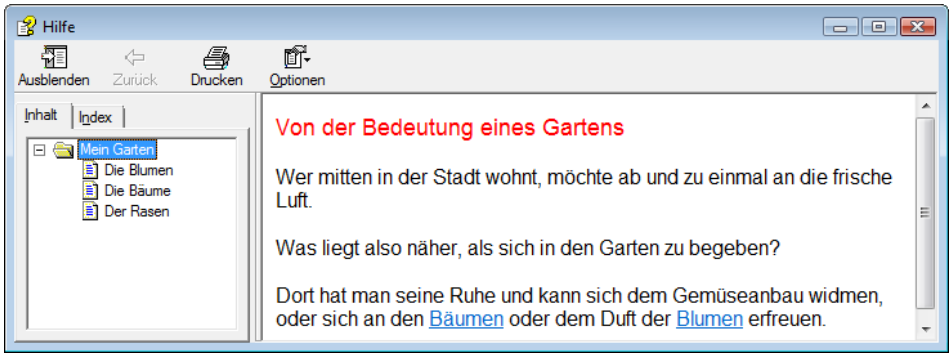


- Wählen Sie "View compiled file" (vorletzte Schaltfläche in der oberen Symbolleiste), so können Sie in einem einzelnen Hilfefenster bereits die erste Hilfeseite sehen und sich über die von Ihnen angelegten Links zu den anderen Seiten bewegen.



- Um das Inhaltsverzeichnis zu erstellen, öffnen Sie die Contents-Seite, bestätigen die Option "Create a new contents file" und speichern es als *Test.hhc* im Projektverzeichnis.
- Klicken Sie auf die Schaltfläche *Insert a heading* (links, zweite von oben).
- Wählen Sie als *Entry Title*: "Mein Garten". Klicken Sie auf die *Edit*-Schaltfläche im Dialogfeld *Path or URL* und stellen Sie oben das Projekt-File *test.hhp* und unten über die *Browse*-Schaltfläche die Seite *allgemein.htm* ein.
- Klicken Sie auf die Schaltfläche *Insert a page* (links, dritte von oben). Das Meldungsfeld "Do you want to insert at the beginning?" quittieren Sie mit *Nein*.
- Weisen Sie auf die gleiche Art die *html*-Dateien der ersten untergeordneten Seite zu.

- Die gleiche Prozedur wiederholen Sie für alle weiteren untergeordneten Seiten. Beginnen Sie dabei immer im Hauptfenster mit der Schaltfläche "Insert a page". verwenden Sie hier keine *Add*-Schaltfläche, sonst kommt es zu einem Compiler-Fehler!
- Nach Verlassen des HTML-Workshops öffnen Sie die Hilfe, indem Sie auf die kompilierte Hilfedatei *test.chm* doppelklicken.



- Das Hinzufügen einer Indexdatei (*Index.hkk*) funktioniert ähnlich wie bei einer Inhaltsdatei. Diesmal öffnen Sie die Indexseite über die Schaltfläche *Insert a keyword*.
- Die einzelnen Keywords ordnen Sie auf analoge Weise (*Edit*-Schaltfläche) den jeweiligen *htm*-Seiten zu.

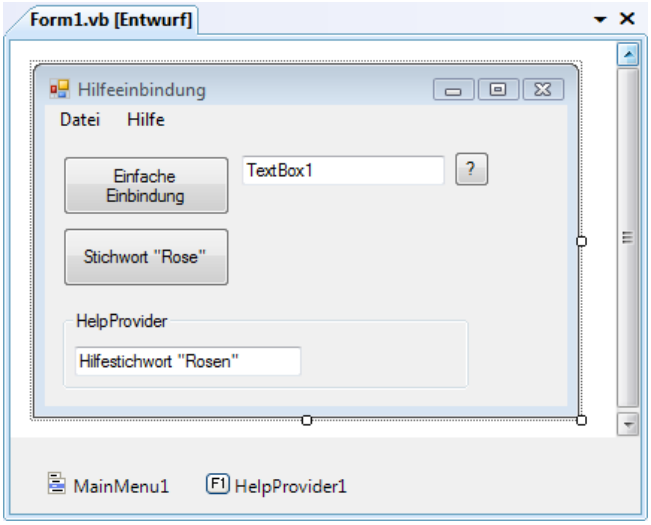
15.3.3 Hilfedateien in die VB-Anwendung einbinden

Altgediente Windows-Programmierer werden zunächst einmal recht frustriert nach der bisher üblichen *HelpContextId*-Eigenschaft Ausschau halten und nichts finden. Die bisher übliche Vorgehensweise wird unter Visual Studio nicht mehr unterstützt. Stattdessen finden Sie zwei neue Controls und ein *Help*-Objekt vor, die alle bisherigen Aufgaben übernehmen.

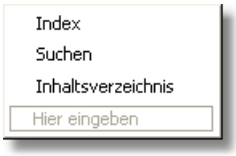
Am Beispiel der Hilfedatei aus dem vorhergehenden Abschnitt möchten wir Ihnen die Vorgehensweise aufzeigen.

Oberfläche

Entwerfen Sie zunächst eine Oberfläche entsprechend folgender Abbildung. Neben den sichtbaren Komponenten fügen Sie noch eine *HelpProvider*- und eine *MainMenu*-Komponente in das *Form* ein.



Erzeugen Sie anschließend ein Hilfemenü mit folgenden Einträgen:



Quelltext (Hilfemenü)

HINWEIS: Kopieren Sie die Hilfedatei aus dem Vorgängerabschnitt in das *\bin\Debug* Verzeichnis der Anwendung.

Mit Hilfe des *Help*-Objekts (Sie können keine Instanzen bilden) ist es möglich, aus der Applikation heraus eine HTML-Hilfedatei oder eine einzelne HTML-Seite aufzurufen. Als Hauptverwendungsbereich dürfte sich der obige Menüpunkt *Hilfe* anbieten, der in keiner professionellen Anwendung fehlen sollte.

Die Routinen, die sich hinter den Menüpunkten *Inhalt*, *Index* und *Suchen* befinden, können Sie mit der *ShowHelp*-Methode realisieren:

Anzeige der Tabulatorseiten *Inhaltsverzeichnis*, *Index*, *Suchen*:

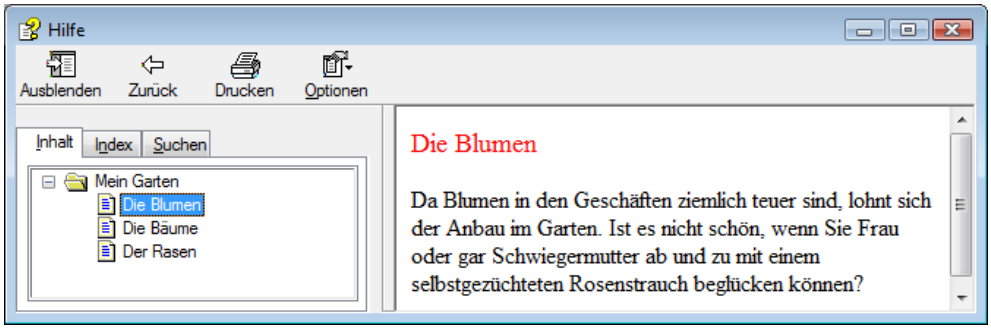
```
Public Class Form1
```

Anzeige des Index mit vorgegebenem Stichwort:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Help.ShowHelp(Me, "garten.chm", HelpNavigator.KeywordIndex, "Rosen")
    End Sub
```

```
    Private Sub MenuItem3_Click(sender As Object, e As EventArgs) Handles MenuItem3.Click
        Help.ShowHelp(Me, "garten.chm", HelpNavigator.Index)
    End Sub

    Private Sub MenuItem4_Click(sender As Object, e As EventArgs) Handles MenuItem4.Click
        Help.ShowHelp(Me, "garten.chm", HelpNavigator.Find)
    End Sub

    Private Sub MenuItem5_Click(sender As Object, e As EventArgs) Handles MenuItem5.Click
        Help.ShowHelp(Me, "garten.chm", HelpNavigator.TableOfContents)
    End Sub

End Class
```

Das Ergebnis nach dem Aufruf:



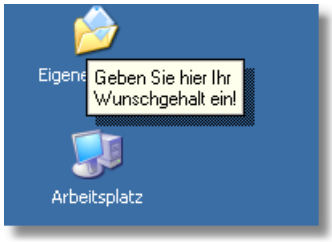
Quelltext (PopUp-Hilfe)

Mit der *ShowPopup*-Methode realisieren Sie relativ problemlos eine einfache Hilfestellung für beliebige Controls, ohne erst eine Hilfedatei programmieren zu müssen.

Anzeige einer PopUp-Hilfe:

```
Help.ShowPopup(Me, "Geben Sie hier Ihr Wunschgehalt ein!", New Point(100, 20))
```

Doch ach, wo landet unser PopUp-Fenster?

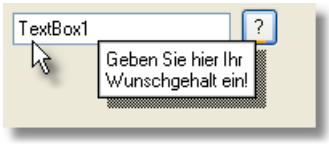


Weit weg vom eigentlichen Geschehen wird die Hilfe angezeigt. Nachbessern ist also angesagt. Dank der neuen Methode *PointToScreen* ist die lästige Koordinatenrechnerei schnell realisiert.

Umrechnen der Anzeigeposition auf Bildschirmkoordinaten:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) _
                          Handles Button3.Click
    Help.ShowPopup(Me, "Geben Sie hier Ihr Wunschgehalt ein!",
                   TextBox1.PointToScreen(New Point(100, 20)))
End Sub
```

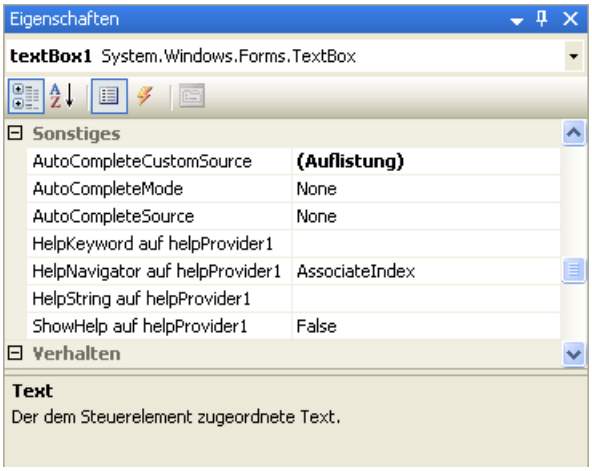
Und jetzt läuft es auch so wie gewünscht:



Quelltext (HelpProvider)

Sicher haben Sie sich auch schon gefragt, wie Sie einzelnen Controls ein bestimmtes Hilfe-Thema zuordnen können. Auch intensivstes Studium der Eigenschaftenliste bringt keine relevanten Erkenntnisse – von Hilfeunterstützung keine Spur. Abhilfe schafft ein recht unscheinbares Control, das zur Laufzeit nicht sichtbar ist:

Mit dem *HelpProvider*-Control aktivieren bzw. realisieren Sie die Hilfeunterstützung für ein Formular. Ziehen Sie einfach die entsprechende Komponente aus dem Werkzeugkasten in ein Windows Form. Bis jetzt nichts Spektakuläres, doch werfen Sie einmal einen Blick auf die Eigenschaften eines beliebigen Eingabe-Controls (z.B. *TextBox*). Unter dem Stichwort *Sonstiges* tauchen plötzlich einige neue Eigenschaften auf, deren Bedeutung unschwer zu erkennen ist:

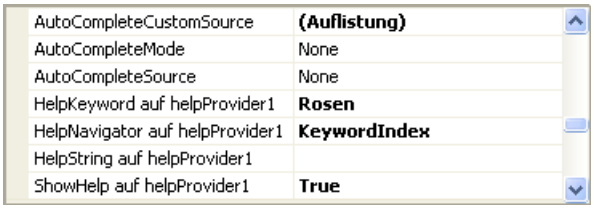


Doch bevor es so weit ist Verknüpfungen herzustellen, sollten Sie die Hilfedatei mit dem *HelpProvider* über dessen *HelpNameSpace*-Eigenschaft (URL) verbinden.

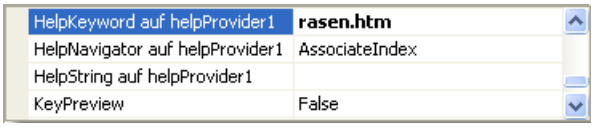
Um die einfache Hilfeunterstützung für das Formular zu aktivieren genügt es, wenn Sie die Formulareigenschaft *Show Help auf HelpProvider* aktivieren (*True*).

Etwas aufwändiger wird es, wenn Sie den einzelnen Eingabe-Controls einen Hilfetopic zuordnen wollen.

Wählen Sie das entsprechende Control (*TextBox2*) und weisen Sie die in der folgenden Abbildung angegebenen Eigenschaften zu:



Alternativ können Sie die einzelnen HTML-Seiten auch über den Namen der HTML-Datei auswählen:



15.3.4 Eine alternative Hilfe-IDE verwenden

Ist Ihnen die bisher beschriebene Hilfeprogrammierung zu kompliziert und zu aufwändig? Wenn ja, dann haben wir nach all den ausufernden Beschreibungen in den Vorgängerabschnitten eine gute Nachricht für Sie. Entweder Sie investieren einige hundert Euro in ein kommerzielles Help-Entwicklungssystem (wie zum Beispiel *HelpStudio* oder *RoboHELP*), oder Sie stecken Ihre Ansprüche etwas zurück und begnügen sich mit dem kostenlosen *HelpMaker*.

Hier die Downloadadresse:

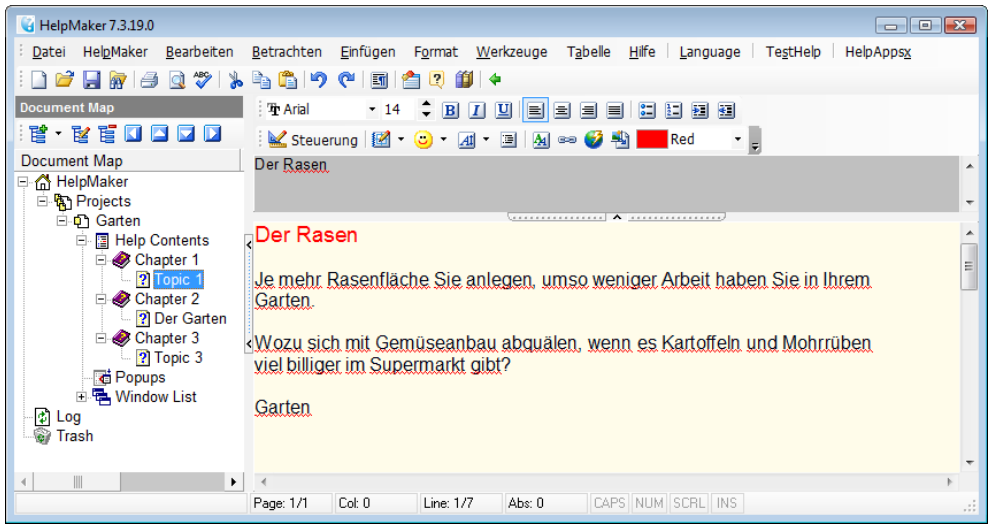
LINK: `http://www.softpedia.com/get/Authoring-tools/Help-e-book-creators/HelpMaker.shtml`

Neben der komfortablen Projektverwaltung nimmt Ihnen das Tool auch jede Menge Arbeit beim Schreiben der einzelnen Hilfeseiten (Keywords etc.) ab. Wie Sie der folgenden Abbildung entnehmen können, ist auch gleich ein kompletter Texteditor in das Programm integriert, Sie brauchen also keine weiteren Tools.

Folgende Ausgabemöglichkeiten bieten sich an:

- HTML-Help,
- WinHelp,

- Browser-Help (eine Website)
- und last but not least eine gedruckte Dokumentation.



Das Tool selbst verwendet intern ebenfalls den Microsoft-Hilfe-Compiler, Sie brauchen sich jedoch weder um dessen Einbindung, noch um dessen Konfiguration zu kümmern, alle Optionen können Sie über die Eigenschaftendialoge bequem einstellen.

Kapitel 16

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,

16.1 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.

16.1.1 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 16.1: Kopieren von Text

VB
```
If TextBox1.SelectedText <> String.Empty Then
    Clipboard.SetText(TextBox1.SelectedText)
Else
    MessageBox.Show("Kein Text selektiert!")
End If
```

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 16.2: Einfügen von Text

VB
```
If Clipboard.ContainsText Then
    TextBox2.Text = Clipboard.GetText
Else
    MessageBox.Show("Keine geeigneten Daten im Clipboard!")
End If
```

BEISPIEL 16.3: Eine Bitmap, von irgendwoher (z.B. aus einem Word-Dokument) in die Zwischenablage kopiert, wird in einer *PictureBox* angezeigt.

VB
```
If Clipboard.ContainsImage Then
    Dim pic As System.Drawing.Image = Clipboard.GetImage
    PictureBox1.Image = pic
Else
    MessageBox.Show("Keine Bitmap im Clipboard!")
End If
```

16.1.2 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 16.4: Kopieren

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    TextBox1.Copy()
End Sub
```

BEISPIEL 16.5: Ausschneiden

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    TextBox1.Cut()
End Sub
```

BEISPIEL 16.6: Einfügen

VB
```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    TextBox1.Paste()
End Sub
```

16.2 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.

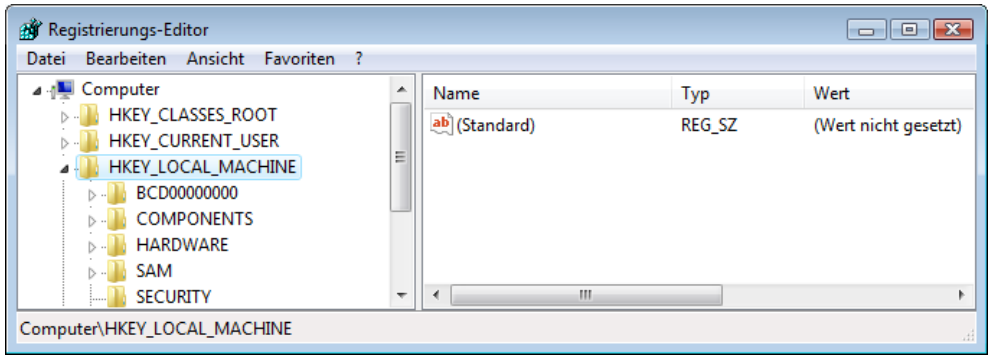
HINWEIS: Vergessen Sie nicht, den Namespace *Microsoft.Win32* in Ihr Programm aufzunehmen, wenn Sie die beiden Klassen nutzen wollen.

16.2.1 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 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.
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 16.7: 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.

16.2.2 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

Eigenschaft	Für den Zugriff auf ...
PerformanceData	HKEY_PERFORMANCE_DATA
Users	HKEY_USERS

BEISPIEL 16.8: Zugriff auf den Baumzweig HKEY_CURRENT_USER

VB
```
Imports Microsoft.Win32
...
    Dim reg As RegistryKey
    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.
OpenSubKey	... öffnet einen Untereintrag.
SetValue	... setzt einen Wert.

Ein Praxisbeispiel zur Registry finden Sie unter

► 16.4.1 Zugriff auf die Registry

16.3 .NET-Reflection

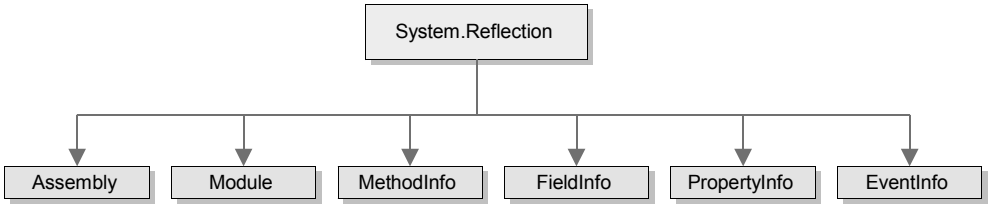
Bei manch hyperaktivem Datenschützer schrillen jetzt wahrscheinlich die Alarmglocken, denn in .NET ist ein perfides "Spitzelsystem" integriert, das die Möglichkeit bietet, jederzeit Informationen zu allen Metadaten einer Anwendung/Assembly zu gewinnen. Man nennt diesen Vorgang auch Reflexion. Die dazu 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.

16.3.1 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 16.3.4, Dynamisches Laden von Assemblies).

Die wichtigsten Klassen auf einen Blick:



16.3.2 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 16.9: Laden einer Assembly

VB

```
Imports System.Reflection
...
    Private myAss As Assembly
    ...
    myAss = Assembly.LoadFrom("xxxxx")...
...
```

Geht es um die aktuell ausgeführte Assembly, können Sie mit *GetExecutingAssembly* die Instanz abrufen:

```
myAss = Assembly.GetExecutingAssembly()
```

16.3.3 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.

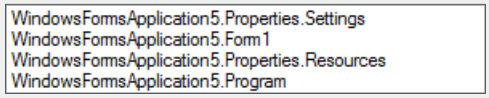
BEISPIEL 16.10: Auflisten aller Typen der aktuellen Assembly

VB

```
    myAss = Assembly.GetExecutingAssembly()
    Dim myType As Type() = myAss.GetTypes()
    For Each tp As Type In myType
        ListBox1.Items.Add(tp.FullName)
    Next
```

Ergebnis

Die Ausgabe:



Alternativ können Sie mit *GetType* auch gezielt einen bestimmten Typ abrufen.

BEISPIEL 16.11: Abrufen der Klasse *Form1* aus der aktuellen Assembly

VB

```
myAss = Assembly.GetExecutingAssembly()
Dim myType As Type = myAss.GetType("WindowsApplication3.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 16.12: Das aktuelle Formular soll näher untersucht werden.

VB

```
    myAss = Assembly.GetExecutingAssembly()
    Dim tp As Type = myAss.GetType("WindowsApplication3.Form1")
    ListBox1.Items.Clear()
```

Name in die *ListBox* eintragen:

```
    ListBox1.Items.Add("      Typ = " & tp.Name.ToString())
    ListBox1.Items.Add("")
```

Falls es sich um eine Klasse handelt:

```
    If tp.IsClass Then
```

Alle Eigenschaften:

```
        For Each p As PropertyInfo In tp.GetProperties()
            ListBox1.Items.Add("  - " & p.Name.ToString())
        Next
        ListBox1.Items.Add("")
```

Alle Methoden:

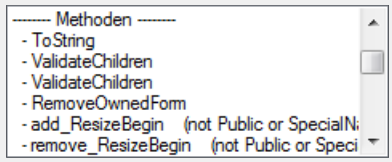
```
        For Each m As MethodInfo In tp.GetMethods()
            If m.IsPublic And Not m.IsSpecialName Then
                ListBox1.Items.Add("  - " & m.Name.ToString())
            Else
                ListBox1.Items.Add("  - " & m.Name.ToString() &
                      "    (not Public or SpecialName)")
            End If
        Next
        ListBox1.Items.Add("")
```

Alle Felder:

```
        For Each f As FieldInfo In tp.GetFields()
            ListBox1.Items.Add("  - " & f.Name.ToString())
        Next
    End If
```

Ergebnis

Die Ausgabe (Ausschnitt):



16.3.4 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.

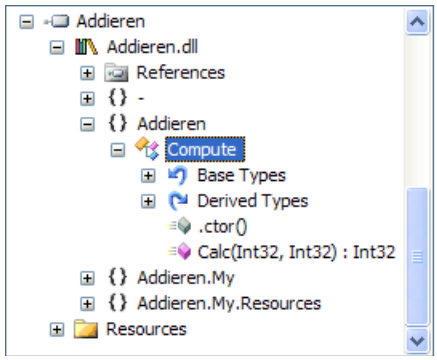
Beispiel

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 folgende, "umfangreiche" Klasse hinzu:

```
Public Class Compute
    Public Function Calc(a As Integer, b As Integer) As Integer
        Return a + b
    End Function
End Class
```

Das Ergebnis im .NET-Reflector:



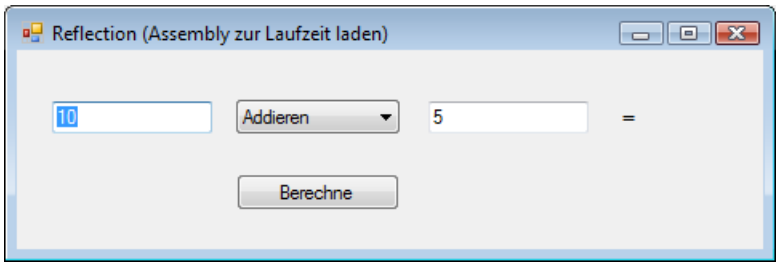
Die Library *Subtrahieren* ist ähnlich einfach aufgebaut:

```
Public Class Compute
    Public Function Calc(a As Integer, b As Integer) As Integer
```

```
        Return a - b
    End Function
End Class
```

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:

```
Imports System.Reflection
Imports System.IO

Public Class Form1
```

Mit dem Laden des Formulars informieren wir uns zunächst, welche Assemblies uns zur Verfügung stehen und tragen deren Namen in die *ComboBox* ein. So können später jederzeit weitere Bibiotheken hinzugefügt werden.

```
    Protected Overrides Sub OnLoad(e As EventArgs)
        Dim myDir As DirectoryInfo
        Dim myFiles As FileInfo()
        myDir = New DirectoryInfo(Application.StartupPath)
        myFiles = myDir.GetFiles("*.dll")
        For Each f As FileInfo In myFiles
            ComboBox1.Items.Add(f.Name.Substring(0, f.Name.Length - 4))
        Next
        ComboBox1.SelectedIndex = 0
        MyBase.OnLoad(e)
    End Sub
```

Mit dem Klick auf den *Button* wird es ernst:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Laden des Typs aus der gewünschten Assembly:

```
        Dim tp As Type = Assembly.LoadFrom(ComboBox1.Text &
                         ".dll").GetType(ComboBox1.Text & ".Compute")
```

Instanz erstellen und Methode abrufen (hier sehen Sie die Vorteile von Reflection):

```
        Dim obj As Object = Activator.CreateInstance(tp)
        Dim meth As MethodInfo = tp.GetMethod("Calc")
```

Die Eingabewerte in *Integer*-Werte umwandeln:

```
        Dim p1 As Integer = Convert.ToInt32(TextBox1.Text)
        Dim p2 As Integer = Convert.ToInt32(TextBox2.Text)
```

Die Methode *Calc* aufrufen und den Rückgabewert auswerten:

```
        Dim p3 As Integer = DirectCast(meth.Invoke(obj, New Object() {p1, p2}), Integer)
        Label1.Text = "= " & p3.ToString()
    End Sub
End Class
```

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.

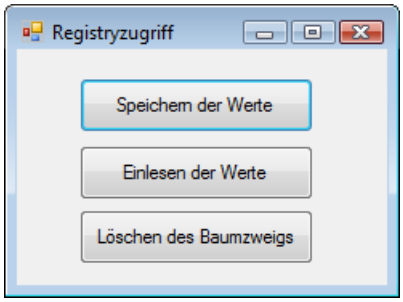
16.4 Praxisbeispiele

16.4.1 Zugriff auf die Registry

Nach all dem theoretischen Geplänkel im Abschnitt 16.2 möchten Sie auch Taten sehen, wenden wir uns also einem kleinen Beispielprogramm zu.

Oberfläche

Lediglich ein Formular und drei *Button*s:



Quelltext

Einbinden des Namespace:

```
Imports Microsoft.Win32
Public Class Form1
```

Das Speichern der Werte:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim reg As RegistryKey, key As RegistryKey
```

Öffnen des Schlüssels *Software*:

```
        reg = Registry.CurrentUser.OpenSubKey("Software", True)
```

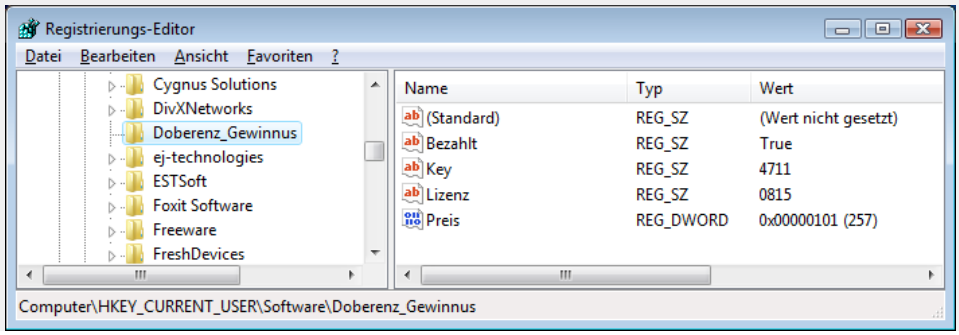
Erzeugen des neuen Untereintrags:

```
        key = reg.CreateSubKey("Doberenz_Gewinnus")
```

Speichern von Werten:

```
        key.SetValue("Lizenz", "0815")
        key.SetValue("Key", "4711")
        key.SetValue("Preis", 257)
        key.SetValue("Bezahlt", True)
    End Sub
```

Das Ergebnis in der Registry:



Einlesen der Werte:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    Dim reg As RegistryKey
    Dim wert As Object
```

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())
End Sub
```

Das Löschen eines Werts:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    Dim reg As RegistryKey
    reg = Registry.CurrentUser.OpenSubKey("Software\Doberenz_Gewinnus", True)
    reg.DeleteValue("Lizenz")
    reg.Close()
```

Das Löschen eines Baumzweigs:

```
    Registry.CurrentUser.OpenSubKey("Software", True).
                        DeleteSubKeyTree("Doberenz_Gewinnus")
End Sub
```

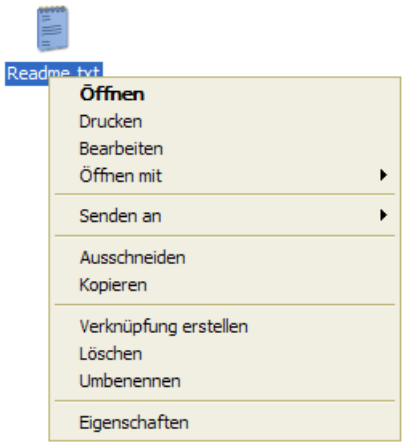
Test

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.

16.4.2 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:

[1] Diese Option ist beim Abfragen von Programmeinstellungen nützlich, am Anfang werden Defaultwerte eingelesen, später kann der Nutzer diese ändern.



Unterstützen können Sie

- *Open* (Öffnen),
- *Print* (Drucken) und
- *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.

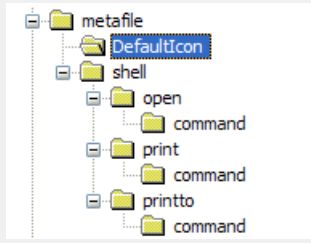
BEISPIEL 16.13: Umsetzung des obigen Beispiels als VB-Programm

VB

```
...
Imports Microsoft.Win32

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim regist As RegistryKey
        Dim key As RegistryKey
        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")
    End Sub
End Class
```

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*).

16.4.3 Die Zwischenablage überwachen und anzeigen

Wer hatte nicht schon einmal das Problem, mehrere Grafiken oder Textabschnitte über die Zwischenablage in eine andere Anwendung zu kopieren? Möchten Sie zum Beispiel eine Grafik aus einer Anwendung ausschneiden und verarbeiten, müssen Sie mehr Tasten drücken, als Ihnen lieb ist.

Das Beispielprogramm soll diesem Missstand abhelfen. Ändert sich der Zwischenablageinhalt, zeigt das Programm die Daten an. Diese könnten Sie zum Beispiel automatisch speichern oder konvertieren. Voraussetzung ist allerdings, dass es sich um Texte (TXT) oder einfache Grafiken (Bitmaps etc.) handelt.

Außerdem erfahren Sie, wie Sie mit Visual Basic Windows-Botschaften abfangen und auswerten können.

Oberfläche

Lediglich ein Windows *Form*, eine *TextBox* sowie eine *PictureBox*.

Quelltext

Das Hauptproblem dieser Anwendung besteht darin, eine Änderung des Zwischenablageinhalts zu registrieren. Eine Polling-Schleife und der dauernde Vergleich mit dem letzten Durchlauf dürften sich aus Performance-Gründen verbieten. Ein Blick in die Windows-API verrät, dass die Änderung des Zwischenablageinhalts mit einer Windows-Botschaft "angezeigt" wird. Zwei Probleme bleiben allerdings bestehen:

- Wie erhalte ich diese Botschaft?
- Wie werte ich sie aus?

Die Lösung des ersten Problems besteht darin, sich in die Kette von Zwischenablagebetrachtern "einzuklinken", da nur diese die Botschaft *WM_DrawClipboard* erhalten. Problem Nummer zwei lösen wir durch das Überschreiben der Fenster-Botschaftsbehandlung (*WndProc*).

```
Imports System.Runtime.InteropServices

Public Class Form1
```

Die beiden API-Konstanten:

```
    Const WM_CHANGECBCHAIN As Integer = 781
    Const WM_DRAWCLIPBOARD As Integer = 776

    <DllImport("user32.dll", CharSet:=CharSet.Auto)>
    Public Shared Function SendMessage(hwnd As IntPtr, wMsg As Integer,
                      wParam As IntPtr, lParam As IntPtr) As Integer
    End Function

    <DllImport("User32.dll", CharSet:=CharSet.Auto)>
    Public Shared Function SetClipboardViewer(hWnd As IntPtr) As IntPtr
    End Function

    <DllImport("User32.dll", CharSet:=CharSet.Auto)>
    Public Shared Function ChangeClipboardChain(hWndRemove As IntPtr,
                                            hWndNewNext As IntPtr) As Boolean
    End Function
```

```
Private NextViewer As IntPtr
```

Nach dem Programmstart ordnen wir das Programm in die Kette der Zwischenablagebetrachter ein und merken uns das Handle des nächsten Viewers:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    NextViewer = SetClipboardViewer(Me.Handle)
End Sub
```

Dieses Handle benötigen wir beim Beenden des Programms, unser Programm muss aus der Kette entfernt werden:

```
Private Sub Form1_FormClosing(sender As Object, e As FormClosingEventArgs) _
                                    Handles MyBase.FormClosing
    ChangeClipboardChain(Me.Handle, NextViewer)
End Sub
```

In diesem Zusammenhang tritt ein weiteres Problem auf, mit dem wir uns herumschlagen müssen: Wir haben zwar das Handle des nächsten Viewers in der Clipboard-Viewer-Kette gespeichert, was passiert aber, wenn sich dieses Fenster zwischenzeitlich von selbst aus der Kette ausklinkt?

In diesem Fall ist für uns die Message WM_CHANGECBCHAIN von Interesse, Übergabeparameter sind die Handle des entfernten und des folgenden Fensters in der Kette.

Botschaftsbehandlung überschreiben:

```
Protected Overrides Sub WndProc(ByRef m As Message)
    Select Case m.Msg
```

Die eigentliche Botschaftsbehandlung an dieser Stelle. Nach Änderung des Zwischenablageinhalts müssen wir prüfen, ob ein unterstütztes Format vorhanden ist. Wenn ja, übernehmen wir dieses in die Komponenten:

```
        Case WM_DRAWCLIPBOARD
            If Clipboard.ContainsText() Then
                TextBox1.Text = Clipboard.GetText()
            End If
            If Clipboard.ContainsImage() Then
                PictureBox1.Image = Clipboard.GetImage()
            End If
```

Sie sollten nicht vergessen, die Botschaft an eventuelle weitere Betrachterfenster weiterzugeben:

```
            SendMessage(NextViewer, m.Msg, m.WParam, m.LParam)
```

Nach dem Test, ob die Message für unsere Anwendung überhaupt interessant ist, wird gegebenenfalls das Handle des neuen Viewers gespeichert. Sollten sich weitere Fenster in der Betrachterkette befinden, reichen wir die Message weiter:

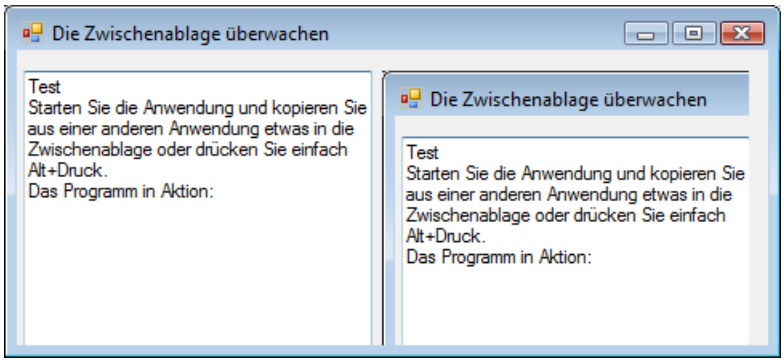
```
        Case WM_CHANGECBCHAIN
            If m.WParam = NextViewer Then
                NextViewer = m.LParam
            Else
```

```
                SendMessage(NextViewer, m.Msg, m.WParam, m.LParam)
            End If
        Case Else
            MyBase.WndProc(m)
    End Select
End Sub
...
End Class
```

Test

Starten Sie die Anwendung und kopieren Sie aus einer anderen Anwendung etwas in die Zwischenablage oder drücken Sie einfach Alt+Druck.

Das Programm in Aktion:



16.4.4 Die WIA-Library kennenlernen

Geht es um den Zugriff auf externe Geräte, wie Scanner, Digitalkameras, Webcams etc., bietet sich dem Programmierer mit der Microsoft WIA-Library (*Windows Image Acquisition Automation Layer*) eine recht komfortable Variante an, deren Vorteil unter anderem darin besteht, ohne viel VBCode auszukommen.

Doch worum geht es eigentlich? Bei WIA handelt es sich um eine komplexe Windows-COM-API, die

- das Auslesen statischer Bilder von Scannern, digitalen Kameras, WebCams,
- das Nachbearbeiten (Größe, Farbe, Drehen ...),
- das Kommentieren (Exif-Informationen)
- und das Konvertieren in diverse Dateiformate (PNG, JPG, GIF, BMP, TIFF)

ermöglicht.

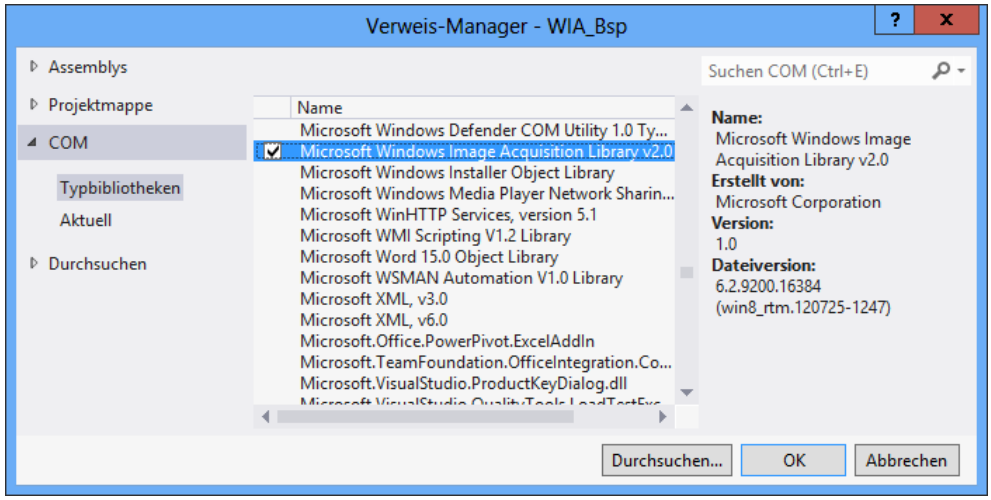
HINWEIS: WIA steht dem Programmierer als COM-Komponente in der Datei *wiaaut.dll* zur Verfügung.

Die Unterstützung beschränkt sich auf die Windows-Version XP ab SP1 und alle später erschienenen Windows Versionen (Vista, 7, 8, Server ...).

Windows Vista ... Windows 10

Die komplette und aktuelle WIA-Libary ist bereits auf Ihrem System installiert, Sie können sofort loslegen, indem Sie die *Microsoft Windows Image Acquisition Library v2.0* in Ihr VB-Projekt einbinden. Erstellen Sie dazu ein neues Projekt und fügen Sie die o.g. Library über *Projekt|Verweis hinzufügen* hinzu (siehe folgende Abbildung).

HINWEIS: Ab Visual Studio 2010 wird die PIA[1] (bzw. die nötigen Datentypen) automatisch in die Anwendung eingebunden, eine Weitergabe der extra Assembly ist also nicht mehr nötig.



Windows XP

Hier ist noch etwas Nacharbeit angesagt, da sich auf Ihrem System derzeit nur die Version 1.x befindet. Laden Sie zunächst die aktuelle Version der WIA von der Microsoft-Webseite:

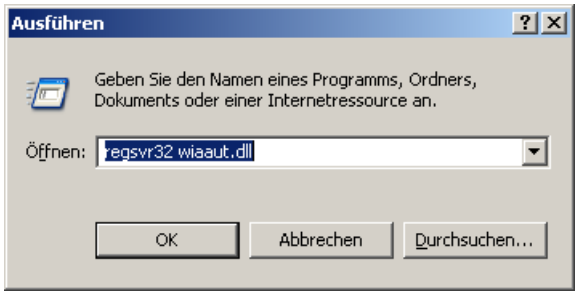
LINK: `http://www.microsoft.com/downloads/details.aspx?FamilyID=a332a77a-01b8-4de6-91c2-b7ea32537e29`

Entpacken Sie die ZIP-Datei und kopieren Sie enthaltene Datei *wiaaut.dll* in das *\\Windows\System32*-Verzeichnis.

[1] *Primary Interop Assembly*

HINWEIS: Ganz nebenbei finden Sie in dieser Datei auch einige VB6-Beispiele und eine umfangreiche Hilfedatei.

Die neue Library registrieren Sie über *Start/Ausführen*:



Binden Sie anschließend die Library, bzw. eine Referenz auf diese (siehe vorhergehender Abschnitt), in Ihr Projekt ein.

Einstieg mit kleiner Beispielanwendung

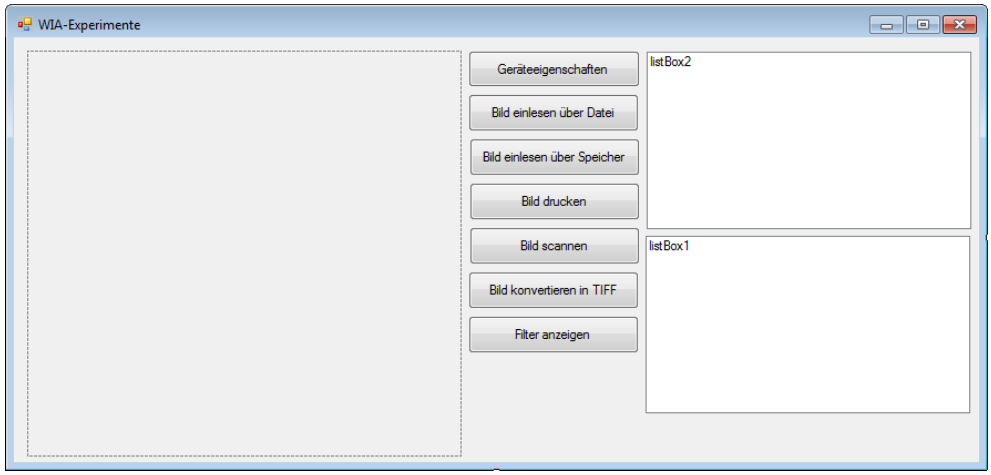
Auf Grund der umfangreichen Möglichkeiten dieser Library möchten wir Ihnen anhand einer kleinen Beispielanwendung einige wesentliche Features, die für das Einlesen und Weiterverarbeiten von Bildern unabdingbar sind, vorstellen:

- Erkennen der angeschlossenen Geräte/Gerätetypen
- Auswerten von Ereignissen (Anschließen/Trennen)
- Anzeige der Geräteeigenschaften
- Einlesen von Bildern
- Verwendung des Scan-Assistenten
- Einsatz des Druckassistenten

Oberfläche

Entwerfen Sie zunächst eine Oberfläche entsprechend der folgenden Abbildung. Fügen Sie dazu zwei *ListBoxen* (Anzeige von Statusmeldungen, Anzeige der aktiven Geräte) sowie ein *PictureBox*-Steuerelement in das Formular ein. Die Bedeutung der Schaltflächen dürfte aus der Beschriftung ersichtlich sein.

Abschließend fügen Sie noch eine *OpenFileDialog*-Komponente ein, mit der wir Bilddateien auswählen wollen.



Quellcode

Ein wichtiger Hinweis vorweg:

HINWEIS: Erzeugen Sie so wenig globale Objekte wie möglich, es besteht immer die Möglichkeit, dass der Anwender die externen Geräte vom PC trennt bzw. diese ausschaltet. In diesem Fall ist die Referenz auf das entsprechende Objekt futsch.

```
Imports System.IO
Imports System.Diagnostics
Imports System.Drawing.Imaging
```

Das zentrale Objekt für den Zugriff auf alle Geräte ist der *DeviceManager*:

```
Public Class Form1
    Dim dm As WIA.DeviceManager
```

Um die PIA-Klassen einbinden zu können, definieren wir einige wichtige Konstanten direkt, andernfalls müssten wir die externe PIA mitgeben:

```
    Const wiaEventDeviceConnected As String = "{A28BBADE-64B6-11D2-A231-00C04FA31809}"
    Const wiaEventDeviceDisconnected As String = "{143E4E83-6497-11D2-A231-00C04FA31809}"
    Const wiaFormatBMP As String = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatPNG As String = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatGIF As String = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatJPEG As String = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatTIFF As String = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}"
```

Mit dem Laden des Formulars wird zunächst der *DeviceMananger* instanziiert, nachfolgend können zwei wichtige Ereignisse angemeldet werden:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        dm = New WIA.DeviceManager()
        dm.RegisterEvent(wiaEventDeviceConnected, "*")
```

```
        dm.RegisterEvent(wiaEventDeviceDisconnected, "*")
        AddHandler dm.OnEvent, AddressOf dm_OnEvent
        anzeige()
    End Sub
```

HINWEIS: Die Verwendung von *RegisterEvent* ist unbedingt erforderlich, andernfalls wird das betreffende Ereignis nicht an die Ereignismethode weitergeleitet.

Reagieren auf das Verbinden/Trennen von Geräten

Nutzen Sie die folgende Ereignisbehandlung, um mit Ihrer Anwendung auf das Hinzufügen bzw. Entfernen von WIA-Geräten zu reagieren:

```
    Private Sub dm_OnEvent(EventID As String, DeviceID As String, ItemID As String)
        If EventID = wiaEventDeviceDisconnected Then
            ListBox1.Items.Add("Gerät getrennt")
        End If
        If EventID = wiaEventDeviceConnected Then ListBox1.Items.Add("Gerät angeschlossen")
        anzeige()
    End Sub
```

HINWEIS: Nur angemeldete Ereignisse werden ausgelöst!

Ermitteln der verfügbaren Geräte

Die folgende Routine aktualisiert die Anzeige der vorhandenen Geräte. Dazu werden alle vom *DeviceManager* in der Collection *DeviceInfos* aufgelisteten Geräte abgefragt und mit *Connect* instanziiert:

```
    Private Sub anzeige()
        Dim dev As WIA.Device
        ListBox2.Items.Clear()
        For Each di As WIA.DeviceInfo In dm.DeviceInfos
            dev = di.Connect()
```

Konnte eine Instanz gebildet werden, wird der Name des Geräts in die Liste eingetragen:

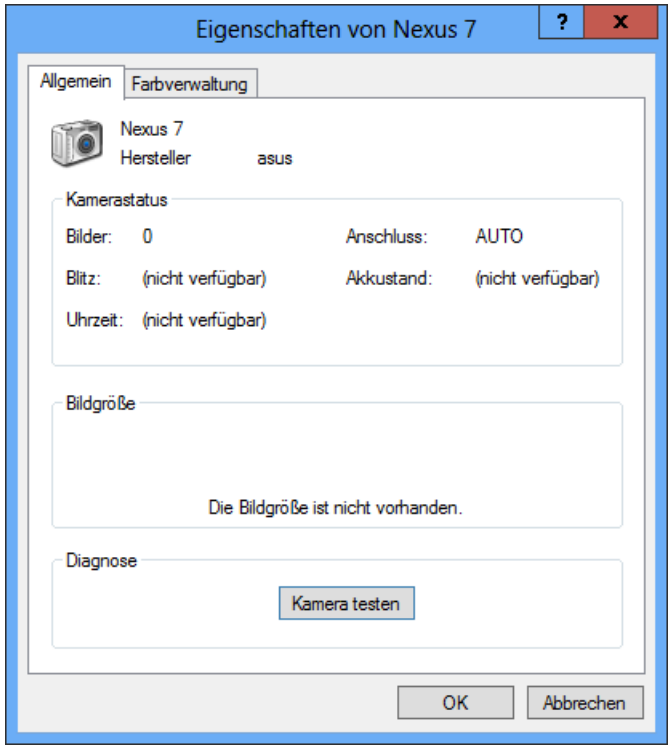
```
            If dev IsNot Nothing Then ListBox2.Items.Add(di.Properties("Name").Value)
        Next di
```

Die Schaltflächen unserer Anwendung werden nur freigegeben, wenn mindestens ein Gerät installiert ist:

```
        Button1.Enabled = ListBox2.Items.Count > 0
        Button2.Enabled = ListBox2.Items.Count > 0
        Button4.Enabled = ListBox2.Items.Count > 0
        Button5.Enabled = ListBox2.Items.Count > 0
        If ListBox2.Items.Count > 0 Then ListBox2.SelectedIndex = 0
    End Sub
```

Anzeige der Geräteeigenschaften

Erfreulicherweise stellt die WIA-Library bereits einen großen Fundus an Dialogen bereit. Unter anderem befindet sich darunter auch ein Dialog zur Anzeige der Geräteeigenschaften, den Sie mit einem *CommonDialog*-Objekt (nicht mit den entsprechenden Dateidialogen verwechseln) und der Methode *ShowDeviceProperties* anzeigen können:




```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim dev As WIA.Device
        Dim dlg As New WIA.CommonDialog()
```

Zunächst wird eine *Device*-Instanz erzeugt (ist dem physischen Gerät zugeordnet):

```
        dev = dm.DeviceInfos(ListBox2.SelectedIndex + 1).Connect()
        dlg.ShowDeviceProperties(dev)
    End Sub
```

Alternativ können Sie auch einzelne Eigenschaften über die *Properties*-Auflistung abrufen:

```
        dev = dm.DeviceInfos(ListBox2.SelectedIndex + 1).Connect()

        For Each prop As WIA.Property In dev.Properties
            ListBox1.Items.Add(prop.Name & ": " & prop.Value)
        Next prop
```

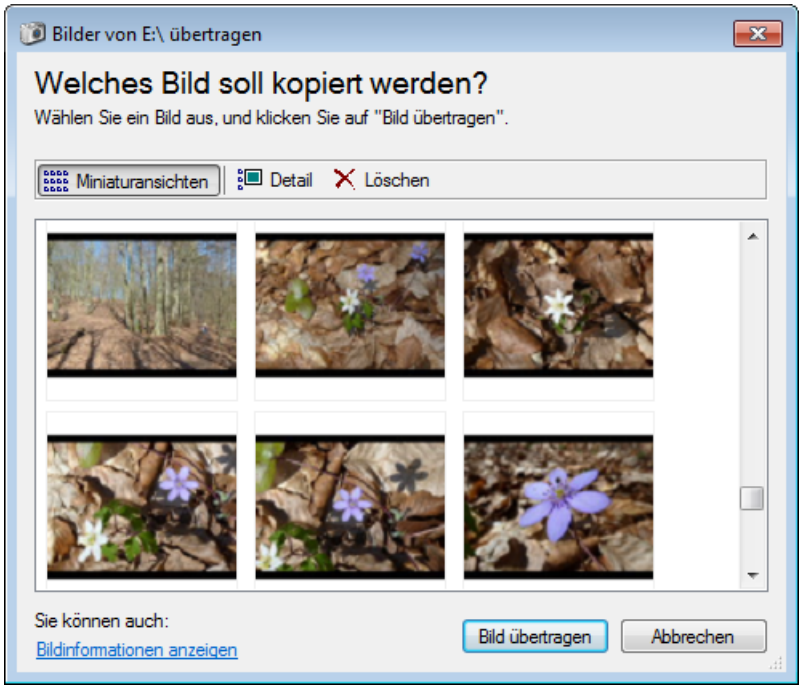
Die Ausgabe:

```
Item Name: Root
Full Item Name: 0018\Root
Item Flags: 76
Unique Device ID: {EEC5AD98-8080-425f-922A-DABF3D
Manufacturer: MATSHITA
Description: E:\
Type: 131072
Port: AUTO
Name: E:\
Server: local
Remote Device ID:
UI Class ID: {00000000-0000-0000-0000-000000000000}
```

Ein Bild einlesen

Das eigentliche Ziel unserer Bemühungen war und ist immer noch das Einlesen eines Bildes, was wir jetzt in Angriff nehmen.

Wie schon beim vorhergehenden Anzeigen von Eigenschaften können wir auch hier auf einen integrierten Dialog zugreifen, mit dem der Anwender das eigentliche Bild auswählt bzw. die Eigenschaften einstellt (siehe folgende Abbildung).



```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim dlg As New WIA.CommonDialog()
        Dim img As WIA.ImageFile
        img = dlg.ShowAcquireImage()
```

Der Rückgabewert ist ein *ImageFile*-Objekt, dessen Dateiformat Sie über die *FileExtension*-Eigenschaft in Erfahrung bringen können.

Für die weitere Vorgehensweise bieten sich zwei Varianten an:

- Sie speichern das Bild als Datei und laden diese gegebenenfalls später.
- Sie übertragen direkt die binären Bilddaten an Ihre Anwendung und nutzen diese beispielsweise für eine *PictureBox*.

An dieser Stelle beschränken wir uns zunächst auf die erste Variante:

```
        Dim TempFileName As String = Path.Combine(Path.GetTempPath(),
                                     Path.GetRandomFileName()) & "." & img.FileExtension
        img.SaveFile(TempFileName)
        Using fs As New FileStream(TempFileName, FileMode.Open, FileAccess.Read)
            pictureBox1.Image = Image.FromStream(fs)
            fs.Close()
        End Using
        File.Delete(TempFileName)
    End Sub
```

HINWEIS: Die Verwendung der *Image.FromFile*-Methode wäre zwar mit weniger Schreibaufwand verbunden, allerdings haben Sie nachfolgend keinen Zugriff auf die Datei und somit können Sie diese temporäre Datei auch nicht löschen.

Alternativ hier der Weg über einen *MemoryStream*:

```
        Dim dlg As New WIA.CommonDialog()
        Dim img As WIA.ImageFile
        img = dlg.ShowAcquireImage()
```

Hier ermitteln wir die Bilddaten,

```
        Dim v As WIA.Vector = img.FileData
```

kopieren diese in ein Byte-Array

```
        Dim bytes() As Byte = CType(v.BinaryData(), Byte())
```

und wandeln dieses in einen *MemoryStream* um:

```
        Dim ms As New MemoryStream(bytes)
```

Letzter Schritt ist die Anzeige des Bildes:

```
        pictureBox1.Image = Image.FromStream(ms)
```

Haben Sie statt einer Digitalkamera einen Scanner angeschlossen, erscheint der bekannte Scan-Dialog von Windows[1].

In diesem Dialog können Sie auch spezielle Einstellungen (Größe, DPI, Farbe) vornehmen.

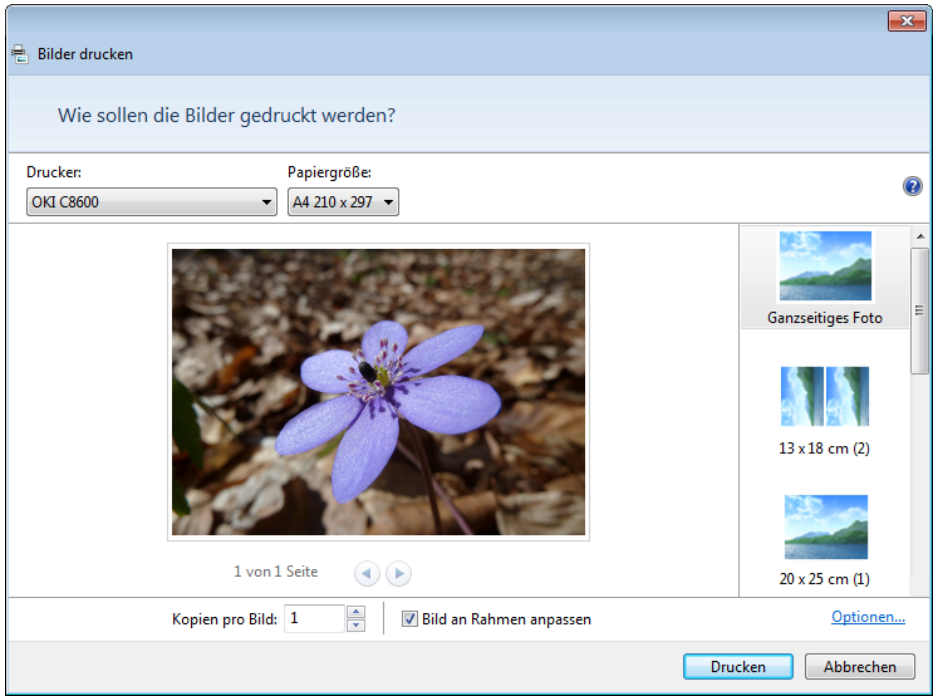
Doch Vorsicht:

HINWEIS: Gerade hier sollten Sie auf den Datentransfer per Arbeitsspeicher verzichten, können doch die Bilder sehr groß werden..

Bild(er) drucken per Assistent

Neben der reinen Anzeige können Sie per WIA auch gleich die Ausgabe realisieren, ohne sich große Gedanken um ein ansprechendes Interface machen zu müssen. Ein entsprechender Assistent steht bereits "ab Werk" zur Verfügung:

[1] Unsere treuen Stammleser werden sich vermutlich noch an unser allererstes Buch (siehe Abbildung) von 1995 erinnern.




```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim dlg As New WIA.CommonDialog()
```

Wird ein Bild per *OpenFileDialog* ausgewählt:

```
        If OpenFileDialog1.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
```

... zeigen wir den Windows-Druckassistenten an:

```
            dlg.ShowPhotoPrintingWizard(OpenFileDialog1.FileName)
        Else
            MessageBox.Show("Kein Bild gewählt!")
        End If
    End Sub
```

Den Scanner-Assistent aufrufen

Nicht genug der Assistenten, auch für das Scannen in eine Datei steht ein entsprechender Assistent bereit:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim dev As WIA.Device
        Dim dlg As New WIA.CommonDialog()
        dev = dm.DeviceInfos(ListBox2.SelectedIndex + 1).Connect()
```

Nur wenn es sich um einen Scanner handelt, wird der Dialog angezeigt:

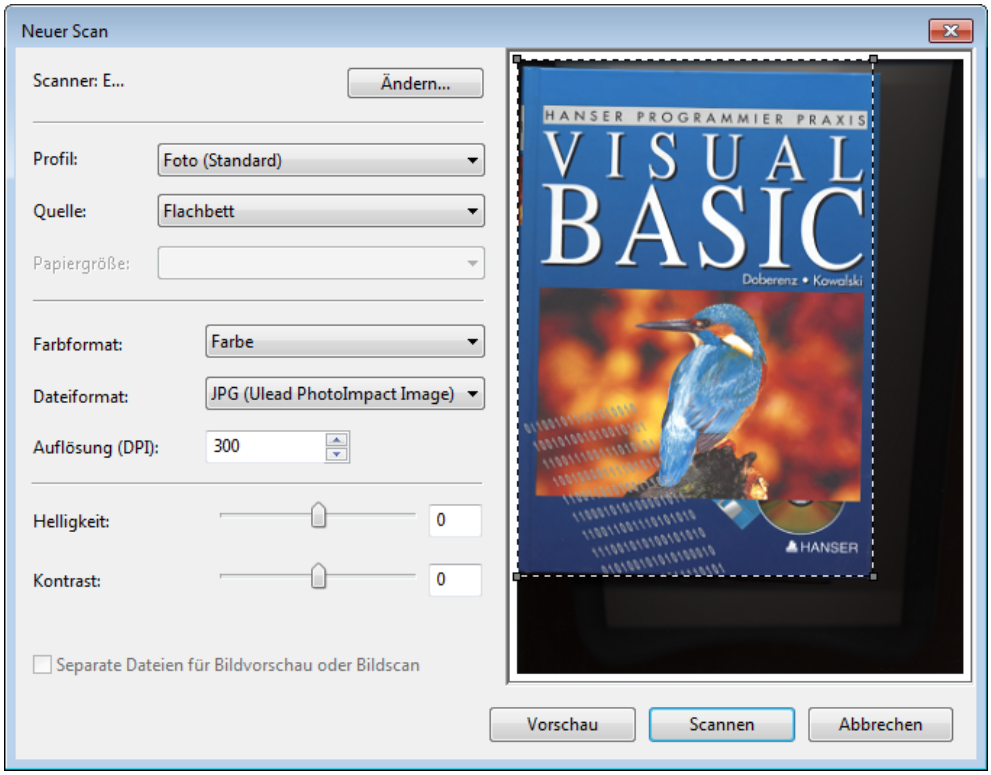
```
        If dev.Type = WIA.WiaDeviceType.ScannerDeviceType Then
```

```
            dlg.ShowAcquisitionWizard(dev)
        Else
            MessageBox.Show("Kein Scanner vorhanden!")
        End If
    End Sub
```

Wie Sie sehen, können Sie per *Device.Type*-Eigenschaft den Gerätetyp in Erfahrung bringen:

Konstante	Bedeutung
ScannerDeviceType	Scanner, nur diese können Sie auch mit dem Scanner-Assistenten verwenden.
CameraDeviceType	Digitale Kameras die per WIA eingebunden sind. Achtung, hier haben Sie es meist mit Verzeichnisstrukturen zu tun!
VideoDeviceType	WebCam etc.
UnspecifiedDeviceType	Unbekannt, wird meist bei der Auswahl von Geräten angegeben, um alle Geräte abzufragen.

Der Scandialog unterscheidet sich in diesem Fall etwas vom *ShowAcquireImage*-Dialog (siehe folgende Abbildung).



Grafikbearbeitung mit WIA

Neben dem reinen Import bietet sich WIA auch für die Bearbeitung von Bildern an. Nachdem diese in einem *ImageFile*-Objekt vorliegen, können diese mit dem *ImageProcess*-Objekt verarbeitet werden. Dazu erstellen Sie zunächst eine Instanz des *ImageProcess*-Objekts und weisen per *Filters.Add*-Methode spezielle Verarbeitungsfilter zu. Anschließend können Sie diese noch konfigurieren.

Konvertieren einer eingescannten Grafik (ins TIFF-Format):

```
        Dim dlg As New WIA.CommonDialog()
        Dim img As New WIA.ImageFile()
        Dim proc As WIA.ImageProcess

        If OpenFileDialog1.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
```

Bild einlesen:

```
            img.LoadFile(OpenFileDialog1.FileName)
```

Konverter erzeugen:

```
            proc = New WIA.ImageProcess()
```

Filter zuweisen:

```
            proc.Filters.Add(proc.FilterInfos("Convert").FilterID)
```

Filter konfigurieren

```
            proc.Filters(1).Properties("FormatID").Value = wiaFormatTIFF
```

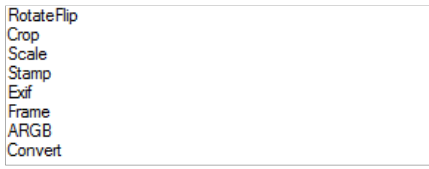
Filter anwenden:

```
            img = proc.Apply(img)
            img.SaveFile("c:\test.tiff")
        End If
    End Sub
```

Verfügbare Filter anzeigen:

```
    Private Sub Button7_Click(sender As Object, e As EventArgs) _
                            Handles Button7.Click
        Dim proc As New WIA.ImageProcess()
        For Each fil As WIA.FilterInfo In proc.FilterInfos
            ListBox1.Items.Add(fil.Name)
        Next fil
    End Sub
```

Namen der verfügbaren Filter:



Natürlich können Sie mit den WIA-Objekten noch weit mehr anfangen, aber für einen ersten Einstieg dürften die bisherigen Ausführungen sicherlich genügen.

16.4.5 Auf eine Webcam zugreifen

Haben Sie schon einmal daran gedacht, Bilder Ihrer Webcam per VB-Programm abzurufen und gegebenenfalls auf einem Server abzulegen? Wenn ja, finden Sie mit Hilfe der Microsoft WIA-Library eine recht einfache Lösung.

Oberfläche

Ein einfaches Windows-Form mit einer *PictureBox* und einem *Timer,* den Sie bitte auf einen nicht zu kleinen Wert einstellen (z.B. 10000 für 10 Sekunden).

HINWEIS: Binden Sie zusätzlich die "Microsoft Windows Image Acquisition Library v2.0" in Ihr VB-Projekt ein (*Projekt|Verweis hinzufügen|COM*).

Quelltext

Binden Sie für die Stream-Unterstützung den folgenden Namespace ein:

```
Imports System.IO

Public Class Form1
```

Die Konstante für die Aufnahme von Bildern:

```
    Private Const wiaCommandTakePicture As String = "{AF933CAC-ACAD-11D2-A093-00C04F72DC3C}"
```

Eine Referenz für das WIA-Gerät, in unserem Fall eine Webcam:

```
    Private Camera As WIA.Device
...
```

Mit dem Laden des Formulars wählen wir zunächst das gewünschte Gerät aus. Sind mehrere Geräte vorhanden, wird ein Auswahldialog angezeigt, sonst wird automatisch das erste Gerät gewählt:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) _
                          Handles MyBase.Load
        Try
            Camera = (New WIA.CommonDialog()).ShowSelectDevice()
```

Testen auf VideoDevice:

```
            If Camera.Type = WIA.WiaDeviceType.VideoDeviceType Then
                timer1.Enabled = True
            Else
                MessageBox.Show("Keine Webcam vorhanden!")
                Application.Exit()
            End If
```

```
        Catch e1 As Exception
            MessageBox.Show("Keine Webcam vorhanden!")
            Application.Exit()
        End Try
    End Sub
```

Ist die Zeit abgelaufen, wird der Schnappschuss ausgelöst:

```
    Private Sub timer1_Tick(sender As Object, e As EventArgs) Handles timer1.Tick
        timer1.Enabled = False
        Me.Text = "ACHTUNG: AUFNAHME"
```

Ein Kommando an das Gerät senden:

```
        Dim item As WIA.Item = Camera.ExecuteCommand(wiaCommandTakePicture)
```

Das erzeugte Bild abrufen und per Speicher an die *PictureBox* weiterleiten:

```
        Dim img As WIA.ImageFile = item.Transfer()
        If img IsNot Nothing Then
            Dim vector As WIA.Vector = img.FileData
            pictureBox1.Image = Image.FromStream(New MemoryStream(CType(
                                                vector.get_BinaryData(), Byte())))
        End If
        Me.Text = "-"
        timer1.Enabled = True
    End Sub
```

Test

Nach dem Start müssen Sie sich einige Sekunden gedulden, dann sollte auch bei Ihnen ein Bild im Fenster erscheinen:

16.4.6 Auf den Scanner zugreifen

Nachdem wir uns im vorhergehenden Beispiel schon etwas mit der WIA beschäftigt haben, wollen wir uns im Rahmen dieses Beispiels noch einmal intensiver mit dem Zugriff auf Scanner befassen.

Zwei grundsätzliche Varianten sind bei Verwendung der WIA möglich:

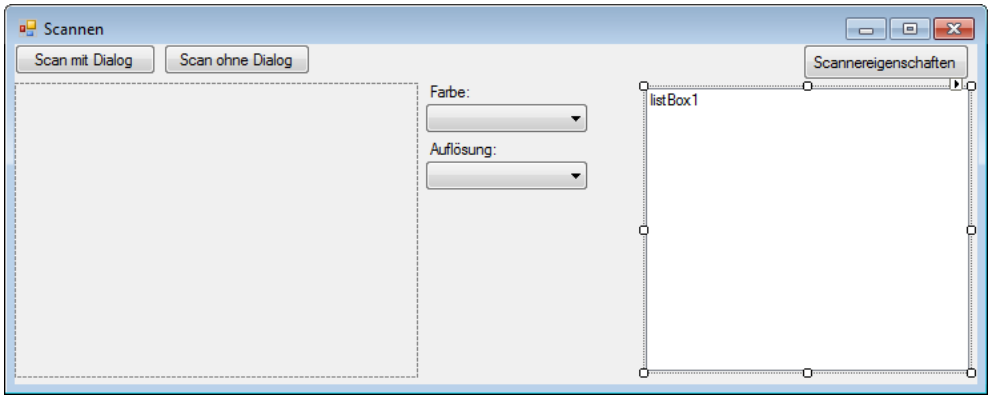
- Anzeige des Scanner-Assistenten und Übernahme des Bildes mit den dort getroffenen Einstellungen.
- Setzen der Scanner-Parameter mit eigenen Dialogen/Komponenten und direktem Abruf des Bildes.

Wie Sie es sich vermutlich denken können, ist die zweite Variante mit wesentlich mehr Programmieraufwand verbunden, müssen Sie sich doch recht intensiv mit den einzelnen Scanner-Parametern auseinandersetzen.

Im Weiteren wollen wir Ihnen beide Versionen vorstellen, zusätzlich beschäftigen wir uns noch mit dem Abruf der Standardeinstellungen.

Oberfläche

Wir beschränken uns auf eine *PictureBox* zur Anzeige, eine *ListBox* für die Scanner-Eigenschaften sowie zwei *ComboBox*en für die Auswahl der Farbeinstellungen und der DPI-Zahl. Last, but not least, sind auch noch drei Schaltflächen nötig:



Weisen Sie der Farb-*ComboBox* die Werte "Schwarz/Weiss", "Graustufe" und "Farbe" zu, die *ComboBox* für die DPI-Zahl konfigurieren Sie mit "75, 100, 150, 200, 300, 600".

HINWEIS: Binden Sie zusätzlich die "Microsoft Windows Image Acquisition Library v2.0" in Ihr VB-Projekt ein (*Projekt|Verweis hinzufügen|COM*).

Quellcode

```
Imports System.IO
Public Class Form1
```

Einige Konstanten für die Auswahl des Bildformats:

```
    Const wiaFormatBMP As String = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatPNG As String = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatGIF As String = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatJPEG As String = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}"
    Const wiaFormatTIFF As String = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}"
...
```

Zunächst beschäftigen wir uns mit der Anzeige des Scannerdialogs:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) _
                                Handles Button1.Click
        Dim dlg As New WIA.CommonDialog()
        Dim img As WIA.ImageFile = Nothing
        img = dlg.ShowAcquireImage(WIA.WiaDeviceType.ScannerDeviceType,
                                   WIA.WiaImageIntent.GrayscaleIntent,
                                   WIA.WiaImageBias.MaximizeQuality, wiaFormatJPEG,
                                   False, False, False)
```

HINWEIS: Machen Sie sich keine Illusionen, es ist (fast) egal, was Sie dem Dialog übergeben, es wird immer die Standardansicht angezeigt.

Wurde ein Bild zurückgegeben, müssen wir dieses an unsere Anwendung weiterreichen:

```
        If img IsNot Nothing Then
            Dim vector As WIA.Vector = img.FileData
            pictureBox1.Image = Image.FromStream(New MemoryStream(CType(vector.BinaryData(),
                                                 Byte())))
        End If
    End Sub
```

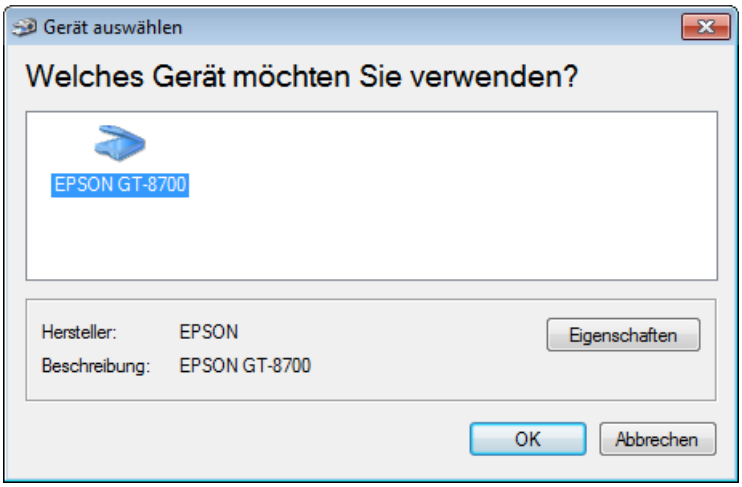
Die etwas flexiblere Variante für den Zugriff auf den Scanner nutzt die WIA-Dialoge lediglich zur Auswahl eines Gerätes. nachfolgend wird das Gerät konfiguriert und es wird das gewünschte Bild abgerufen. Doch der Reihe nach ...

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) _
                              Handles Button2.Click
```

Ein Gerät abrufen:

```
        Dim dlg As New WIA.CommonDialog()
        Dim Scanner As WIA.Device = dlg.ShowSelectDevice(WIA.WiaDeviceType.ScannerDeviceType,
                                   False, False)
```

Der angezeigte Dialog:



Aus der zweiten *ComboBox* lesen wir die DPI-Zahl ab:

```
Dim dpi As Integer = Convert.ToInt16(comboBox2.SelectedItem)
```

Je nach Auswahl in der ersten *ComboBox* weisen wir die Farbauswahl zu:

```
Select Case comboBox1.SelectedIndex
    Case 0
        Scanner.Items(1).Properties("Current Intent").Value = 4
       ' 4 = SW, 2 = Graustufe, 1 = Color
    Case 1
        Scanner.Items(1).Properties("Current Intent").Value = 2
    Case Else
        Scanner.Items(1).Properties("Current Intent").Value = 1
End Select
```

Übergeben wird diese Information über die *Properties*-Collection des ersten Items. Gleiches trifft auch auf die anderen Scan-Optionen zu:

Dots per inch/Horizontal:

```
Scanner.Items(1).Properties("Horizontal Resolution").Value = dpi
```

Dots per inch/Vertikal:

```
Scanner.Items(1).Properties("Vertical Resolution").Value = dpi
```

Linke obere Ecke für den Scanbereich:

```
Scanner.Items(1).Properties("Horizontal Start Position").Value = 0 ' x links oben
Scanner.Items(1).Properties("Vertical Start Position").Value = 0 ' y links oben
```

Größe des Scanbereichs (Inch * dpi):

```
        Scanner.Items(1).Properties("Horizontal Extent").Value = 8.5 * dpi
        Scanner.Items(1).Properties("Vertical Extent").Value = 11 * dpi
```

Übertragen des Bildes in ein *ImageFile*-Objekt:

```
        Dim img As WIA.ImageFile = Scanner.Items(1).Transfer()
```

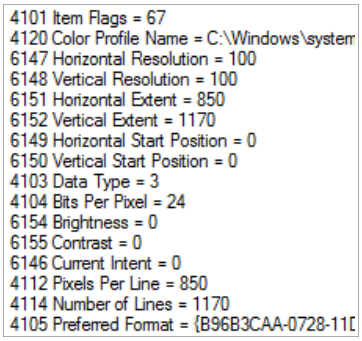
War dies erfolgreich, müssen wir das *ImageFile*-Objekt "nur noch" in unserer *PictureBox* anzeigen:

```
        If img IsNot Nothing Then
            Dim vector As WIA.Vector = img.FileData
            pictureBox1.Image = Image.FromStream(New MemoryStream(
                                CType(vector.BinaryData(), Byte())))
        End If
    End Sub
```

Ganz nebenbei zeigen wir in der *ListBox* noch die möglichen Eigenschaften des Scanners an:

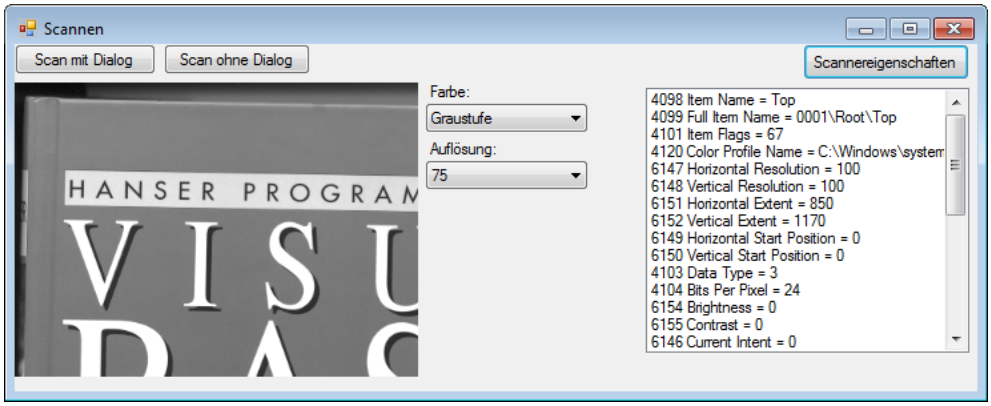
```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Dim dlg As New WIA.CommonDialog()
        Dim Scanner As WIA.Device = dlg.ShowSelectDevice(WIA.WiaDeviceType.ScannerDeviceType,
                                                         False, False)
        For Each p As WIA.IProperty In Scanner.Items(1).Properties
            ListBox1.Items.Add(p.PropertyID & " " & p.Name & " = " & p.Value)
        Next p
    End Sub
End Class
```

Das mögliche Ergebnis (geräteabhängig):



Test

Nach dem Start können Sie beide Scan-Varianten ausprobieren. Experimentieren Sie ruhig auch einmal mit den Properties des *Item*-Objekts um andere Auflösungen oder Scanbereiche zu bestimmen.



Bemerkung

Die Parameter für *ShowAcquireImage*:

```
SYNTAX:  ImageFile ShowAcquireImage([DeviceType], [Intent], [Bias], [FormatID],
                                    [AlwaysSelectDevice], [UseCommonUI], [CancelError])
```

Die Bedeutung:

Parameter	Beschreibung
DeviceType	Werte der *WiaDeviceType*-Enumeration, Default = *UnspecifiedDeviceType*, für Scanner nutzen Sie bitte *ScannerDeviceType*.
Intent	Werte der *WiaImageIntent*-Enumeration (Wert egal)
Bias	Werte der *WiaImageBias*-Enumeration (Wert egal)
FormatID	Das zurückgegebene Bildformat: wiaFormatBMP = "{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}" wiaFormatPNG = "{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}" wiaFormatGIF = "{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}" wiaFormatJPEG = "{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}" wiaFormatTIFF = "{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}"
AlwaysSelectDevice	Soll das Gerät jedes Mal ausgewählt werden?
UseCommonUI	Soll der Dialog angezeigt werden (das ist immer der Fall)?
CancelError	Soll ein Fehler ausgelöst werden, wenn der Nutzer Abbruch wählt?

16.4.7 OpenOffice.org Writer per OLE steuern

Angesichts der reichlich vorhandenen Lösungen zur MS Word-Automatisierung entsteht für manchen Programmierer sicher auch die Frage, wie sich Open*Office.org Writer*, der Freeware-Pendant von Word, automatisieren lässt. Das vorliegende Beispiel soll einen ersten Ansatz aufzeigen.

Leider ist die Programmierung von OpenOffice-Anwendungen per C#/Visual Basic nicht gerade trivial, eine fast schon chaotisch wirkende Programmierschnittstelle macht es dem eingefleischten MS-Programmierer nicht gerade leicht.

Wir wollen Ihnen etwas Arbeit abnehmen und kapseln deshalb einige der wichtigsten Grundfunktionen in einer eigenen Klasse *OOWriter*.[1] Mit dieser lassen sich dann die Standardaufgaben relativ leicht lösen.

HINWEIS: Um die Wiederverwendbarkeit zu verbessern, haben wir die Funktionalität in eine eigene Library ausgelagert, Sie können die enthaltene Klasse *cOOWriter* aber auch genausogut in Ihr Projekt kopieren[2].

VB-Library "OOWLib"

Erstellen Sie zunächst ein Windows Forms-Projekt und fügen Sie in die übergeordnete Solution eine Klassenbibliothek mit dem Namen *OOWLib* ein (Kontextmenü *Hinzufügen neues Projekt*).

Löschen Sie den schon vorhandenen Klassenrumpf und erstellen Sie eine neue Klasse *cOOWriter*, auf deren Details wir im Folgenden eingehen wollen.

Quelltext (Klasse cOOWriter)

Unsere neue Klasse:

```
Public Class cOOWriter
```

Einige interne Objekte:

```
    Private oStarOffice As Object
    Private oDesk As Object
    Private oDoc As Object
    Private oVC As Object
```

Wir erstellen ein neues, leeres Dokument:

```
    Public Sub NewDocument()
        Dim args(-1) As Object
        oDesk = oStarOffice.createInstance("com.sun.star.frame.Desktop")
```

Das aktuelle Dokument:

```
        oDoc = oDesk.LoadComponentFromURL("private:factory/swriter", "_blank", 0, args)
```

Der sichtbare Cursor[3]:

```
        oVC = oDoc.CurrentController.ViewCursor()
```

[1] Sollte ein Leser den Elan aufbringen, die Klasse wesentlich zu erweitern, sind die Autoren für jede Art von Feedback dankbar.

[2] Ganz nebenbei profitiert von dieser Library auch unsere C#-Leserschaft, da sich VB beim Zugriff auf COM-Objekte nach wie vor als überlegen erweist.

[3] Writer kennt auch noch Textcursor, die unabhängig vom sichtbaren Cursor agieren können.

```
End Sub
```

Alternativ greifen wir auf eine vorhandene Datei zu:

```
Public Sub OpenDocument(doc As String, Optional ro As Boolean = False,
                        Optional Passwort As String = "")
    Dim args(2) As Object
    args(0) = MakePropertyValue("ReadOnly", ro)
    args(1) = MakePropertyValue("Password", Passwort)
    args(2) = MakePropertyValue("Hidden", False)
    oDesk = oStarOffice.createInstance("com.sun.star.frame.Desktop")
    oDoc = oDesk.LoadComponentFromURL(ConvertToUrl(doc), "_blank", 0, args)
    oVC = oDoc.CurrentController.ViewCursor()
End Sub
```

HINWEIS: Sie müssen eine der beiden o.g. Methoden ausführen, bevor Sie das Dokument bearbeiten können.

Ausgabe von Text:

```
Public Sub InsertString(str As String)
    oDoc.Text.InsertString(oVC, str, False)
End Sub
```

Ein Zeilenumbruch:

```
Public Sub NewLine()
    oDoc.Text.InsertControlCharacter(oVC, 0, False)
End Sub
```

Ein neuer Absatz:

```
Public Sub NewParagraph()
    oDoc.Text.InsertControlCharacter(oVC, 5, False)
End Sub
```

Ein Seitenumbruch:

```
Public Sub PageBreak()
    oDoc.Text.InsertControlCharacter(oVC, 0, False)
    oVC.BreakType = 4
End Sub
```

Eine Dokumentvorlage nutzen:

```
Public Sub SetStyle(StyleName As String)
    oVC.ParaStylename = StyleName
End Sub
```

Fettschrift:

```
Public Property Bold() As Boolean
    Get
        Return (oVC.CharWeight = 150)
```

```
        End Get
        Set(value As Boolean)
            If value Then
                oVC.CharWeight = 150
            Else
                oVC.CharWeight = 100
            End If
        End Set
    End Property
```

Eine Schriftart auswählen:

```
    Public Sub SetFont(Fontname As String, Optional Size As Long = 12)
        oVC.CharFontname = Fontname
        If Size > 0 Then oVC.CharHeight = Size
    End Sub
```

An den Anfang des Dokuments springen:

```
    Public Sub GotoStart()
        oVC.JumpToFirstPage()
    End Sub
```

An das Ende des Dokuments springen:

```
    Public Sub GotoEnd()
        oVC.JumpToLastPage()
        oVC.jumpToEndOfPage()
    End Sub
```

Datei als PDF sichern:

```
    Public Sub SaveAsPDF(filename As String)
        Dim args(1) As Object
        args(0) = MakePropertyValue("FilterName", "writer_pdf_Export")
        oDoc.storeToURL(ConvertToUrl(filename), args)
    End Sub
```

Dokument schließen:

```
    Public Sub CloseDocument()
        oDoc.Close(True)
    End Sub
```

Die folgenden internen Funktionen stammen aus dem Internet[1] und erleichtern den Zugriff auf die OpenOffice-Objekte:

Eine Objekt-Eigenschaft erzeugen:

```
    Private Function MakePropertyValue(cName, uValue) As Object
        Dim oPropertyValue As Object
```

[1] http://www.kalitech.fr/clients/doc/VB_APIOOo_en.html

```
        oPropertyValue = oStarOffice.Bridge_GetStruct("com.sun.star.beans.PropertyValue")
        oPropertyValue.Name = cName
        oPropertyValue.value = uValue
        MakePropertyValue = oPropertyValue
    End Function
```

Pfadangaben in Urls umwandeln:

```
    Private Function ConvertToUrl(strFile As String) As String
        strFile = Replace(strFile, "\", "/")
        strFile = Replace(strFile, ":", "|")
        strFile = Replace(strFile, " ", "%20")
        strFile = "file:///" + strFile
        ConvertToUrl = strFile
    End Function
```

Ein spezielles Service-Objekt erzeugen:

```
    Private Function CreateUnoService(strServiceName) As Object
        Dim oServiceManager As Object
        oServiceManager = CreateObject("com.sun.star.ServiceManager")
        CreateUnoService = oServiceManager.createInstance(strServiceName)
    End Function
```

Im Konstruktor erstellen wir das zentrale *ServiceManager*-Objekt:

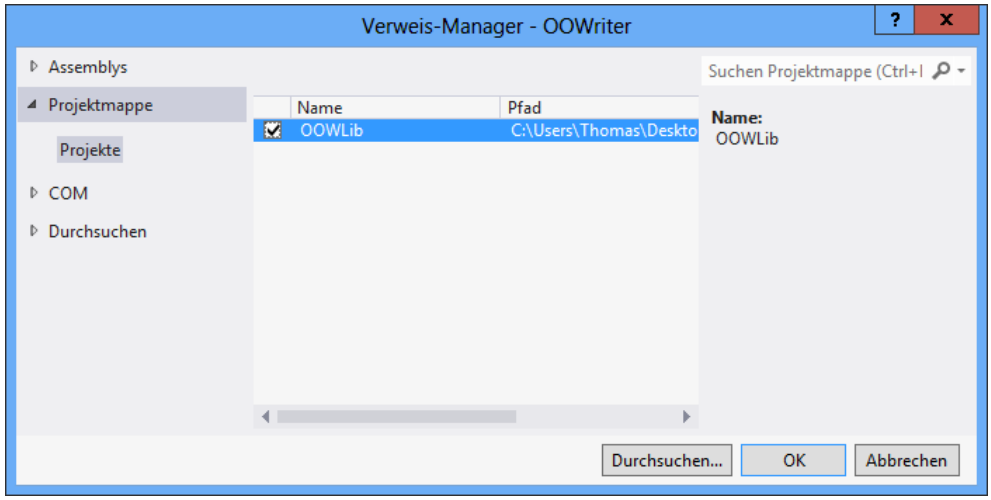
```
    Public Sub New()
        oStarOffice = CreateObject("com.sun.star.ServiceManager")
    End Sub
End Class
```

HINWEIS: Kompilieren Sie abschließend die Klassenbibliothek um die gewünschte Assembly zu erstellen.

Damit endet auch schon unser kurzer Ausflug in die Library-Programmierung und wir kehren wieder zu unserer vertrauten Programmiersprache zurück.

Einbinden der Assembly

Wählen Sie in der Solution das Windows Forms-Projekt und fügen Sie eine Referenz auf die oben erstellte Klassenbibliothek hinzu (*Verweis hinzufügen|Projekte*).



Nachfolgend kommen wir schon in den Genuss der vollen Funktionalität unserer Assembly, wir müssen den Namespace nur noch importieren.

Oberfläche

Die Oberfläche beschränkt sich auf einen *Button* und einen *FileSaveDialog*.

Quelltext

Um das Beispiel nicht auf ein simples "Hello World" zu beschränken, wollen wir den Inhalt einer XML-Datei formatiert ausgeben.

HINWEIS: Die dazu nötige XML-Datei finden Sie in den Beispieldaten zum Buch.

Import der beiden benötigten Namespaces:

```
Imports OOWLib
Imports System.Xml.Linq
Public Class Form1
```

Eine Instanz für die Elemente der XML-Datei:

```
    Private ArtikelListe As XElement

    Public Sub New()
        InitializeComponent()
```

Hier laden wir die XML-Beispieldaten:

```
        ArtikelListe = XElement.Load("Artikel.xml")
    End Sub
```

Mit dem Klick auf die Schaltfläche beginnen die hektischen Aktivitäten:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Eine Instanz des Writers erstellen[1]:

```
Dim oow As New cOOWriter()
```

Ein neues Dokument anlegen:

```
oow.NewDocument()
```

Formatvorlage *Überschrift1* auswählen:

```
oow.SetStyle("Überschrift 1")
```

Textausgabe:

```
oow.InsertString("Lagerdaten")
```

Neuer Absatz und Formatvorlage *Standard:*

```
oow.NewParagraph()
oow.SetStyle("Standard")
```

Zeilenvorschub:

```
oow.NewLine()
```

Hier folgt die Schleife für alle *Artikel*-Elemente in den XML-Daten:

```
For Each Artikel As XElement In ArtikelListe.Elements()
```

Fettschrift und Textausgabe:

```
    oow.Bold = True
    oow.InsertString("Artikelnr : ")
    oow.Bold = False
    oow.InsertString(Artikel.Element("ArtikelNr").Value)
    oow.NewLine()
    oow.Bold = True
    oow.InsertString("Artikelname : ")
    oow.Bold = False
    oow.InsertString(Artikel.Element("Artikelname").Value)
    oow.NewLine()
    oow.Bold = True
    oow.InsertString("Lagerbestand : ")
    oow.Bold = False
    oow.InsertString(Artikel.Element("Lagerbestand").Value)
    oow.NewLine()
    oow.NewLine()
Next Artikel
```

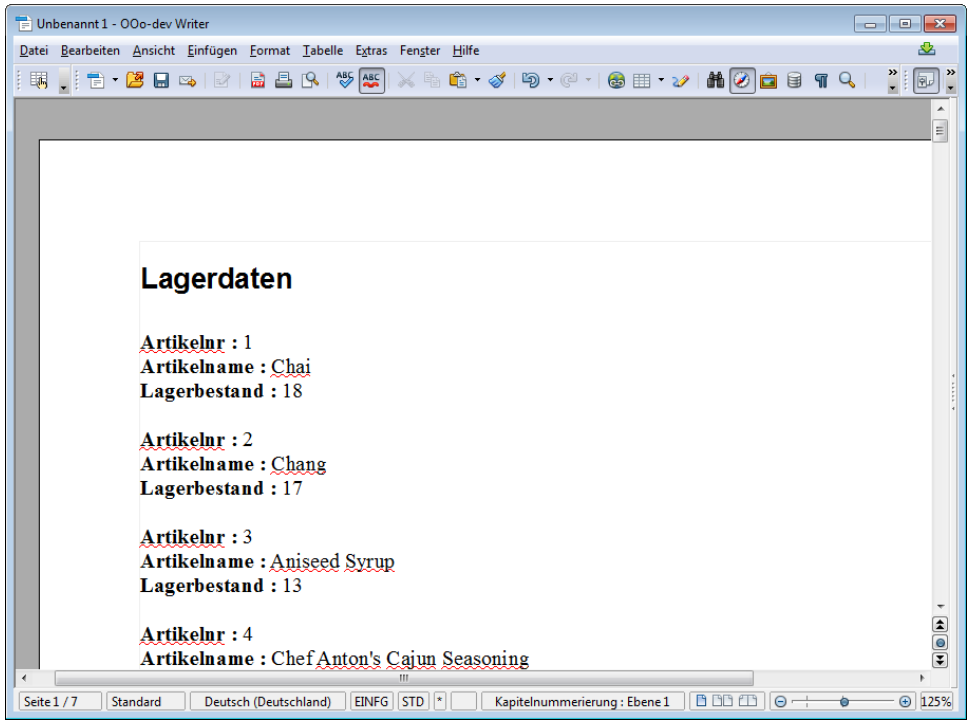
[1] Im Gegensatz zu Word/Excel etc. wird der Writer standardmäßig eingeblendet, Sie müssen diesen also nicht erst sichtbar machen.

Last, but not least wollen wir das erzeugte Writer-Dokument als PDF-Datei speichern:

```
        If SaveFileDialog1.ShowDialog() = System.Windows.Forms.DialogResult.OK Then
            oow.SaveAsPDF(SaveFileDialog1.FileName)
        End If
        oow = Nothing
    End Sub
End Class
```

Test

Nach dem Klick auf die Schaltfläche sollten Sie folgende Ausgabe erhalten (vorausgesetzt, Sie haben *OpenOffice.org* installiert):



Auch der Export ins PDF-Format dürfte problemlos realisiert werden, Sie können sich mit dem Acrobat Reader vom Erfolg überzeugen.

Bemerkung

Auch wenn Sie mit obigem Beispiel recht schnell zu brauchbaren Ergebnissen kommen, viele OpenOffice-Funktionen sind nur umständlich erreichbar (z.B. Zwischenablage).

Hilfe finden Sie in diesem Fall unter:

LINK: `http://forum.openoffice.org/en/forum/`

16.4.8 Nutzer und Gruppen des Systems ermitteln

Dieses Beispiel zeigt Ihnen, wie Sie mit Hilfe der WMIs alle Nutzer und Gruppen des aktuellen Computers ermitteln können.

Oberfläche

Wir brauchen ein Windows *Form* und eine *TreeView*-Komponente (siehe Laufzeitansicht).

HINWEIS: Binden Sie über den Projektmappen-Explorer einen Verweis auf die Assembly *System.Management* ein.

Quelltext

Einbinden der Namespaces:

```
Imports System.Management
Imports System.Net

Public Class Form1
```

Mit dem Laden des Formulars erzeugen wir zunächst die beiden Haupt-Knoten in der *TreeView*:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Dim nUser As TreeNode = TreeView1.Nodes.Add("Nutzer")
        Dim nGroups As TreeNode = TreeView1.Nodes.Add("Gruppen")
```

Eine Abfrage für die WMIs zusammenbasteln:

```
        Dim Query As New ManagementObjectSearcher("SELECT * FROM" &
                        " Win32_UserAccount WHERE Domain='" & Dns.GetHostName() & "'")
```

Ausgabe der Nutzer:

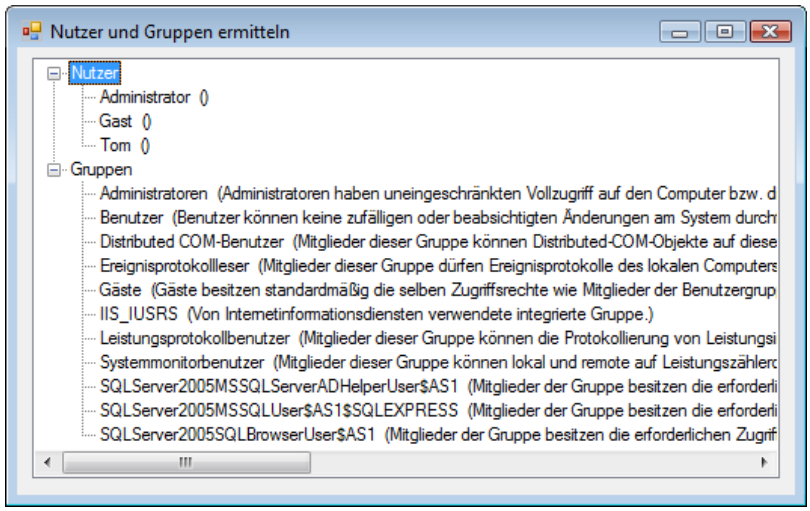
```
        For Each mo As ManagementObject In Query.Get()
            nUser.Nodes.Add(mo("Name").ToString() & "  (" & mo("FullName").ToString() & ")")
        Next
```

Jetzt noch die Gruppen abfragen:

```
        Query = New ManagementObjectSearcher("SELECT * FROM Win32_Group WHERE " &
                                             "Domain='" & Dns.GetHostName() & "'")
        For Each mo As ManagementObject In Query.Get()
            nGroups.Nodes.Add(mo("Name").ToString() & "  (" &
                                mo("Description").ToString() & ")")
        Next
    End Sub

End Class
```

Test

Nach dem Programmstart werden bereits alle Nutzer und Gruppen im Baum angezeigt:



16.4.9 Testen, ob Nutzer in einer Gruppe enthalten ist

Möchten Sie kontrollieren, ob der Nutzer angemeldet ist und zu bestimmten Gruppen gehört, können Sie ein *WindowsIdentity*-Objekt einsetzen.

Oberfläche

Erforderlich sind ein Windows *Form* und eine *ListBox* (siehe Laufzeitansicht).

Quelltext

```
Imports System.Security.Principal

Public Class Form1
```

Mit dem Laden des Formulars rufen wir zunächst die aktuelle Windows Identität ab:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        Dim wi As WindowsIdentity = WindowsIdentity.GetCurrent()
```

Auswerten der Anmeldung:

```
        ListBox1.Items.Add("Username: " & wi.Name)
        ListBox1.Items.Add("IsAuthenticated: " & wi.IsAuthenticated.ToString)
        ListBox1.Items.Add("AuthenticationType: " & wi.AuthenticationType.ToString)
```

Test auf einige vorgegebene Gruppen:

```
        ListBox1.Items.Add("IsAnonymous: " & wi.IsAnonymous.ToString)
        ListBox1.Items.Add("IsGuest: " & wi.IsGuest.ToString)
        ListBox1.Items.Add("IsSystem: " & wi.IsSystem.ToString)
```

Und so können Sie auf eine spezifische Gruppe testen:

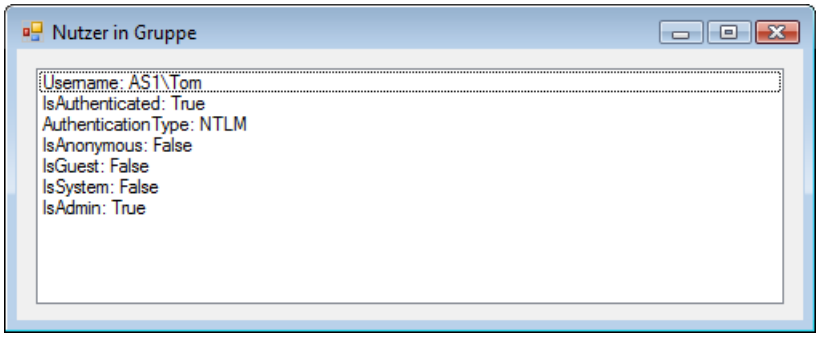
```
        ListBox1.Items.Add("IsAdmin: " & (New WindowsPrincipal(WindowsIdentity.GetCurrent() _
                           ).IsInRole(WindowsBuiltInRole.Administrator)).ToString)
    End Sub
End Class
```

Siehe zu diesem Thema auch das Beispiel

► 16.4.10 Testen, ob der Nutzer ein Administrator ist

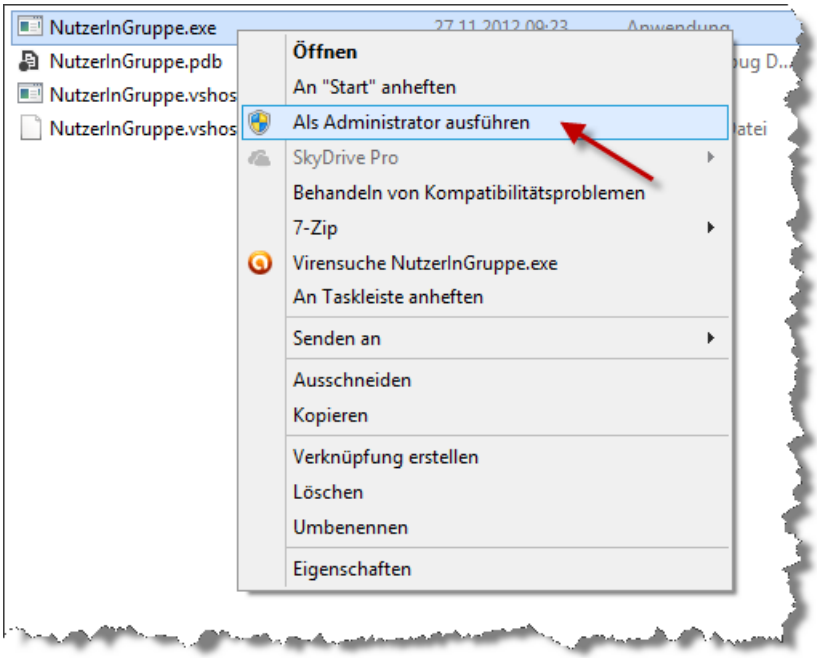
Test

Nach dem Start finden Sie die Informationen in der *ListBox*:



Bemerkung

Falls Sie beim Test festgestellt haben, dass Sie nicht als Administrator identifiziert wurden, denken Sie bitte daran, dass seit Windows Vista die Benutzerkontensteuerung (UAC) darüber entscheidet, mit welchem Zugriffstoken Sie gerade Aufgaben ausführen. Im Normalfall dürfte dies die Standardbenutzer-Rolle sein, haben Sie sich vorher Administratorrechte geholt, wird auch die Administrator-Rolle verwendet (siehe folgende Abbildung):



16.4.10 Testen, ob der Nutzer ein Administrator ist

Für den Zugriff auf einige Systemfunktionen bzw. Änderungen an den Systemeinstellungen sind häufig Administratorenrechte erforderlich. Wie Sie prüfen, ob die aktuelle Anwendung mit Admin-Rechten arbeitet, zeigt das vorliegende Beispiel.

Oberfläche

Nur ein Windows *Form* und ein *Label*.

Quelltext

Binden Sie zunächst den folgenden Namespace ein:

```
Imports System.Security.Principal

Public Class Form1
```

Die kleine Hilfsfunktion:

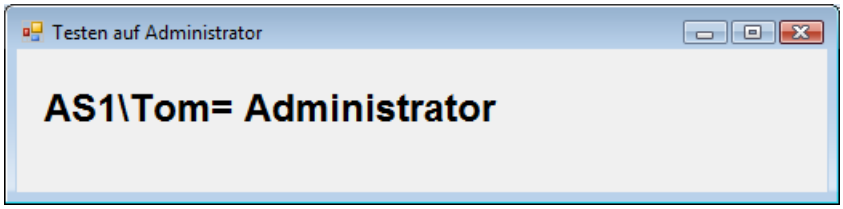
```
    Public Function IsAdmin() As Boolean
        Return (New WindowsPrincipal(WindowsIdentity.GetCurrent() _
                                    ).IsInRole(WindowsBuiltInRole.Administrator))
    End Function
```

Die Verwendung:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        If IsAdmin() Then
            Label1.Text = WindowsIdentity.GetCurrent().Name & " = Administrator"
        Else
            Label1.Text = WindowsIdentity.GetCurrent().Name & " kein Admin"
        End If
    End Sub

End Class
```

Test

Nach dem Start werden Ihnen der aktuelle Nutzer und das Funktionsergebnis angezeigt:



Bemerkung

Denken Sie an die Benutzerkontensteuerung, die Ihnen im Normalfall nur Standardbenutzer-Rechte einräumt. Weitere Details dazu, siehe vorhergehendes Beispiel

► 16.4.9 Testen, ob Nutzer in einer Gruppe enthalten ist

16.4.11 Die IP-Adressen des Computers bestimmen

Soll Ihr Programm über rudimentäre Netzwerkfähigkeiten (Remoting etc.) verfügen, ist meist auch die aktuelle IP-Adresse des Systems von Interesse. Ein kleines Beispiel zeigt, wie es geht.

Oberfläche

Wir brauchen ein Windows *Form* mit einer *ListBox*.

Quelltext

```
Imports System.Net

Public Class Form1
```

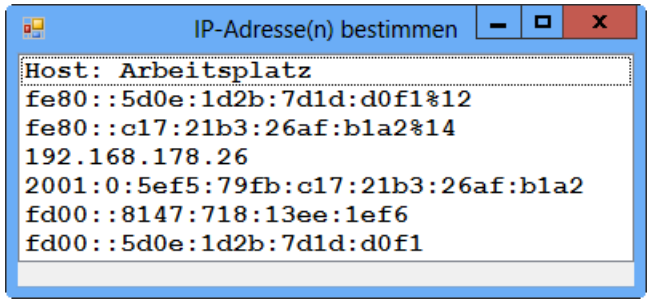
Mit dem Laden des Formulars füllen wir auch die *ListBox* mit allen gefundenen IP-Adressen:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        ListBox1.Items.Add("Host: " & Dns.GetHostName())
```

```
        For Each ip As IPAddress In Dns.GetHostEntry(Dns.GetHostName()).AddressList
            ListBox1.Items.Add(ip)
        Next
    End Sub
End Class
```

Test

Sie haben es sicher schon geahnt, ein Computer kann mehr als eine IP-Adresse haben und so ist es auch im folgenden Beispiel:



Neben dem "normalen" Netzwerkzugang finden sich noch eine DVB-Sat-Karte und eine weitere Netzwerkkarte im Computer.

HINWEIS: Den aktuellen PC können Sie auch über die IP-Adresse 127.0.0.1 erreichen!

16.4.12 Die IP-Adresse über den Hostnamen bestimmen

Während der Nutzer meist mit Hostnamen (z.B. *Server*) statt IP-Adressen arbeitet, sind viele Programmfunktionen nur mit den entsprechenden IP-Adressen realisierbar. In diesem Fall benötigen Sie eine Funktion, die den Hostnamen in eine IP-Adresse umwandelt.

Oberfläche

Lediglich ein Windows *Form* und eine *ListBox* werden benötigt.

Quelltext

Mit dem Laden des Formulars bestimmen wir die IP-Adresse(n) für den Hostname des aktuellen Computers.

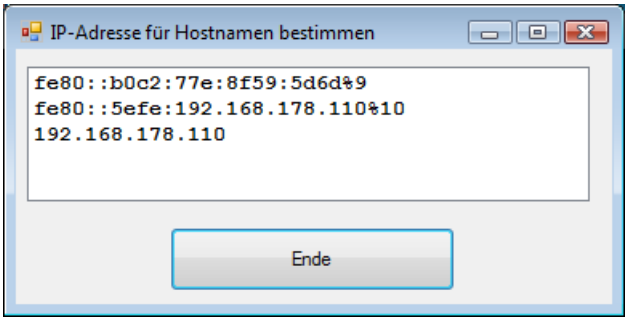
```
Imports System.Net

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        For Each ip As IPAddress In Dns.GetHostEntry(Dns.GetHostName()).AddressList
```

```
            ListBox1.Items.Add(ip.ToString())
        Next
    End Sub
End Class
```

HINWEIS: Sie können natürlich auch den Namen eines anderen Netzwerk-Clients angeben.

Test

Die Programmausgabe:



16.4.13 Diverse Systeminformationen ermitteln

Wer sich eingehend mit der Umgebung des Programms (Betriebssystem, Hardware etc.) beschäftigen will, findet die gesuchten Informationen im Wesentlichen über die folgenden Klassen:

- *SystemInformation*
- *System.Environment*
- *Application*
- *Windows Management Instrumentations* kurz WMI.

Wir wollen im Folgenden die wichtigsten Informationen auslesen, einen Anspruch auf Vollständigkeit erheben wir lieber nicht.

Oberfläche

Ein Windows *Form*, eine *ListBox* und ein *MenuStrip* (siehe Laufzeitansicht).

Binden Sie zusätzlich den Verweise auf *System.Management* in Ihr Projekt ein.

Quelltext

HINWEIS: Um die Übersicht in diesem Beispiel zu erhöhen und ein schnelles Nachschlagen zu erleichtern, wurden zusätzliche Zwischenüberschriften eingefügt.

Einbinden der benötigten Namespaces:

```
Imports System.Collections
Imports System.Management
Imports Microsoft.VisualBasic.Devices
Public Class Form1
```

Betriebssystem (Name, Version, Bootmode)

Über die *Environment*-Klasse stehen die gewünschten Informationen zur Verfügung:

```
    Private Sub dToolStripMenuItem_Click(sender As Object,
                    e As EventArgs) Handles dToolStripMenuItem.Click
        ListBox1.Items.Add("OSVersion: " & Environment.OSVersion.ToString())
        ListBox1.Items.Add("OSVersion.Platform: " & Environment.OSVersion.Platform)
        ListBox1.Items.Add("OSVersion.ServicePack: " & Environment.OSVersion.ServicePack)
        ListBox1.Items.Add("OSVersion.Version: " & Environment.OSVersion.VersionString)
        ListBox1.Items.Add("OSFullName: " & (New ComputerInfo()).OSFullName)
        ListBox1.Items.Add("CurrentCulture: " & Application.CurrentCulture.ToString())
        ListBox1.Items.Add("Bootmode: " & SystemInformation.BootMode.ToString())
    End Sub
```

Ausgabe der obigen Anweisungen:

```
OSVersion: Microsoft Windows NT 6.2.9200.0
OSVersion.Platform: Win32NT
OSVersion.ServicePack:
OSVersion.Version: Microsoft Windows NT 6.2.9200.0
OSFullName: Microsoft Windows 8 Pro
CurrentCulture: de-DE
Bootmode: Normal
```

Wie Sie sehen, bietet nur die *OSFullName*-Eigenschaft den allgemein bekannten Betriebssystemnamen.

Meist genügt jedoch die Abfrage von *Platform*, wenn Sie zwischen *Windows XP* (ja, das gibt es teilweise noch) und einer der NT-Versionen (dies schließt *Vista/7/8* mit ein) unterscheiden wollen.

Auslesen der Registry in Abhängigkeit vom Betriebssystem:

```
        Select Case Environment.OSVersion.Platform
            Case PlatformID.Win32Windows
                Dim rk As RegistryKey = Registry.LocalMachine.OpenSubKey(
                                        "Software\Microsoft\Windows\CurrentVersion", False)
...
            Case PlatformID.Win32NT
                Dim rk As RegistryKey = Registry.LocalMachine.OpenSubKey(
                                       "Software\Microsoft\Windows NT\CurrentVersion", False)
...
        End Select
```

Schriftarten/-Informationen

Allgemeine Informationen über die Darstellung der Schriftarten (*Systemsteuerung|Anzeige|Darstellung*):

```
Private Sub dToolStripMenuItem1_Click(sender As Object,
            e As EventArgs) Handles dToolStripMenuItem1.Click
    ListBox1.Items.Add("IsFontSmoothingEnabled: " &
                       SystemInformation.IsFontSmoothingEnabled.ToString)
    ListBox1.Items.Add("FontSmoothingType: " &
                       SystemInformation.FontSmoothingType.ToString &
                       "  (1=Default, 2=ClearType)")
    ListBox1.Items.Add("FontSmoothingContrast: " &
                        SystemInformation.FontSmoothingContrast.ToString)
```

Die Liste der möglichen Schriftarten abrufen:

```
    For Each ff As FontFamily In FontFamily.Families
        ListBox1.Items.Add(ff.Name)
    Next
End Sub
```

Ausgabe der obigen Anweisungen:

```
IsFontSmoothingEnabled: True
FontSmoothingType: 2  (1=Default, 2=ClearType)
FontSmoothingContrast: 1200
Academy Engraved LET
Aharoni
Amienne
Andalus
Angsana New
AngsanaUPC
Arabic Typesetting
Arial
Arial Black
Arial Narrow
Arnprior
```

Die möglichen Werte für *FontSmoothingType*:

Wert	Bedeutung
1	Standardschriftglättung.
2	ClearType-Schriftglättung.

Bildschirme

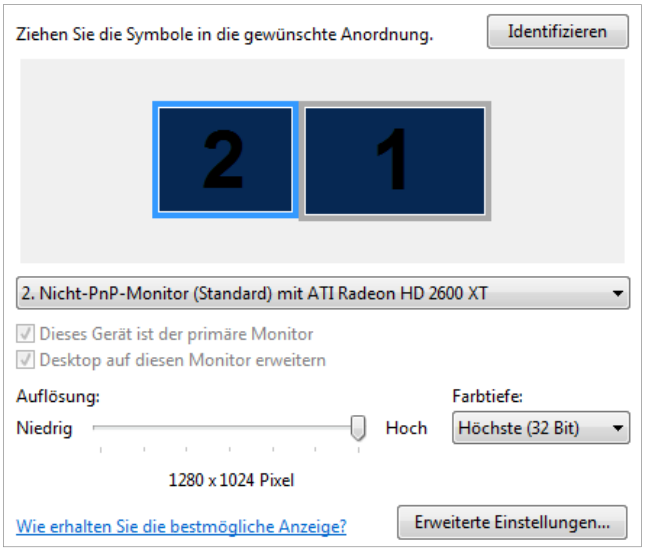
Dass mehr als ein Bildschirm möglich ist, hat sich sicher schon herumgesprochen. Das .NET-Framework bietet geeignete Möglichkeiten, die User-Konfiguration zu bestimmen, auch wenn es nicht ganz einfach ist, den Zusammenhang richtig zu interpretieren:

```
    Private Sub dToolStripMenuItem2_Click(sender As Object, e As EventArgs) _
                                          Handles dToolStripMenuItem2.Click
        ListBox1.Items.Add("MonitorCount: " & SystemInformation.MonitorCount.ToString)
        ListBox1.Items.Add("MonitorsSameDisplayFormat: " &
                           SystemInformation.MonitorsSameDisplayFormat.ToString)
        ListBox1.Items.Add("PrimaryMonitorMaximizedWindowSize: " &
                           SystemInformation.PrimaryMonitorMaximizedWindowSize.ToString)
        ListBox1.Items.Add("PrimaryMonitorSize: " &
                           SystemInformation.PrimaryMonitorSize.ToString)
        ListBox1.Items.Add("ScreenOrientation: " &
                           SystemInformation.ScreenOrientation.ToString)
        ListBox1.Items.Add("VirtualScreen: " & SystemInformation.VirtualScreen.ToString)
        ListBox1.Items.Add("WorkingArea: " & SystemInformation.WorkingArea.ToString)
    End Sub
```

Ausgabe der obigen Anweisungen:

```
MonitorCount: 2
MonitorsSameDisplayFormat: True
PrimaryMonitorMaximizedWindowSize: {Width=1296, Height=1010}
PrimaryMonitorSize: {Width=1280, Height=1024}
ScreenOrientation: Angle0
VirtualScreen: {X=0,Y=0,Width=2960,Height=1050}
WorkingArea: {X=0,Y=0,Width=1280,Height=994}
```

Die oben ermittelten Informationen entsprechen folgender System-Konfiguration:



Screen2: 1280 x 1024 Pixel, *Screen2* 1680 x 1050 Pixel

Netzwerk (User-Name, PC-Name ...)

Auch die wichtigsten Netzwerkinformationen lassen sich über die .NET-Klassen abrufen:

```
Private Sub dToolStripMenuItem3_Click(sender As Object, e As EventArgs) _
                                      Handles dToolStripMenuItem3.Click
    ListBox1.Items.Add("ComputerName: " & SystemInformation.ComputerName.ToString)
    ListBox1.Items.Add("MachineName: " & Environment.MachineName)
    ListBox1.Items.Add("Network: " & SystemInformation.Network.ToString)
    ListBox1.Items.Add("TerminalServerSession: " &
                       SystemInformation.TerminalServerSession.ToString)
    ListBox1.Items.Add("UserName: " & SystemInformation.UserName)
    ListBox1.Items.Add("UserDomainName: " & SystemInformation.UserDomainName)
    ListBox1.Items.Add("UserInteractive: " & SystemInformation.UserInteractive.ToString)
End Sub
```

Ausgabe der obigen Anweisungen:

```
ComputerName: AS1
MachineName: AS1
Network: True
TerminalServerSession: False
UserName: Tom
UserDomainName: AS1
UserInteractive: True
```

Weitere Informationen über Nutzer und Gruppen, siehe Beispiel

► 16.4.9 Testen, ob Nutzer in einer Gruppe enthalten ist

Environment Variablen auslesen

Nicht alle, aber viele Informationen über Betriebssystem etc. können Sie über die Environment-Variablen auslesen:

```
Private Sub dToolStripMenuItem5_Click(sender As Object,
                    e As EventArgs) Handles dToolStripMenuItem5.Click
    For Each de As DictionaryEntry In Environment.GetEnvironmentVariables()
        ListBox1.Items.Add(de.Key & " = " & de.Value)
    Next
End Sub
```

Ausgabe:

```
Path = C:\Program Files\CodeGear\RAD Studio\5.0\bin;C:\Users\Public\Documen
TEMP = C:\Users\Tom\AppData\Local\Temp
BDSCOMMONDIR = C:\Users\Public\Documents\RAD Studio\5.0
PATHEXT = .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
USERDOMAIN = AS1
PROCESSOR_ARCHITECTURE = x86
SystemDrive = C:
RoxioCentral = C:\Program Files\Common Files\Roxio Shared\9.0\Roxio Central
APPDATA = C:\Users\Tom\AppData\Roaming
windir = C:\Windows
LOCALAPPDATA = C:\Users\Tom\AppData\Local
SESSIONNAME = Console
TMP = C:\Users\Tom\AppData\Local\Temp
USERPROFILE = C:\Users\Tom
ProgramFiles = C:\Program Files
FP_NO_HOST_CHECK = NO
HOMEPATH = \Users\Tom
COMPUTERNAME = AS1
USERNAME = Tom
```

Energiestatus

Nicht jeder Computer ist ständig "an der Leine", und so wird auch die Information über den Batteriestatus nicht ganz uninteressant sein:

```
Private Sub dToolStripMenuItem4_Click(sender As Object,
                    e As EventArgs) Handles dToolStripMenuItem4.Click
    ListBox1.Items.Add("PowerStatus.BatteryChargeStatus: " &
                    SystemInformation.PowerStatus.BatteryChargeStatus.ToString())
    ListBox1.Items.Add("PowerStatus.BatteryFullLifetime: " &
                    SystemInformation.PowerStatus.BatteryFullLifetime.ToString())
    ListBox1.Items.Add("PowerStatus.BatteryLifePercent: " &
                    SystemInformation.PowerStatus.BatteryLifePercent.ToString())
    ListBox1.Items.Add("PowerStatus.BatteryLifeRemaining: " &
                    SystemInformation.PowerStatus.BatteryLifeRemaining.ToString())
    ListBox1.Items.Add("PowerStatus.PowerLineStatus: " &
                    SystemInformation.PowerStatus.PowerLineStatus.ToString())
End Sub
```

PC mit Netzversorgung:

```
PowerStatus.BatteryChargeStatus: NoSystemBattery
PowerStatus.BatteryFullLifetime: -1
PowerStatus.BatteryLifePercent: 1
PowerStatus.BatteryLifeRemaining: -1
PowerStatus.PowerLineStatus: Online
```

Hardware-Informationen

Möchten Sie einen Blick unter die "Motorhaube" Ihres PCs werfen, ohne gleich den Schraubenzieher zu bemühen, geht auch dies:

```
Private Sub dToolStripMenuItem6_Click(sender As Object,
                        e As EventArgs) Handles dToolStripMenuItem6.Click
```

Abfrage der Prozessoranzahl:

```
ListBox1.Items.Add("ProcessorCount: " & Environment.ProcessorCount.ToString())
```

Mit Hilfe der WMIs erfahren wir noch etwas mehr über die Prozessoren:

```
Dim Query As New SelectQuery("Win32_Processor")
Dim mos As New ManagementObjectSearcher(Query)
For Each mo As ManagementObject In mos.Get()
    ListBox1.Items.Add("Manufacturer: " & mo("Manufacturer").ToString())
    ListBox1.Items.Add("Name: " & mo("Name").ToString())
    ListBox1.Items.Add("SocketDesignation: " & mo("SocketDesignation").ToString())
    ListBox1.Items.Add("ExtClock: " & mo("ExtClock").ToString())
    ListBox1.Items.Add("Description: " & mo("Description").ToString())
    ListBox1.Items.Add("AddressWidth: " & mo("AddressWidth").ToString())
    ListBox1.Items.Add("CurrentClockSpeed: " & mo("CurrentClockSpeed").ToString() &
                       "MHz")
Next
mos.Dispose()
```

Speicherinfos sammeln:

```
Dim ci As New ComputerInfo()
ListBox1.Items.Add("AvailableVirtualMemory: " & ci.AvailableVirtualMemory.ToString())
ListBox1.Items.Add("AvailablePhysicalMemory: " & ci.AvailablePhysicalMemory)
ListBox1.Items.Add("TotalVirtualMemory: " & ci.TotalVirtualMemory)
ListBox1.Items.Add("TotalPhysicalMemory: " & ci.TotalPhysicalMemory)
End Sub
```

Die Ausgabe:

```
ProcessorCount: 8
Manufacturer: GenuineIntel
Name: Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz
SocketDesignation: LGA1155
ExtClock: 100
Description: Intel64 Family 6 Model 42 Stepping 7
AddressWidth: 64
CurrentClockSpeed: 3501MHz
AvailableVirtualMemory: 8795426705408
AvailablePhysicalMemory: 14027591680
TotalVirtualMemory: 8796092891136
TotalPhysicalMemory: 17145397248
```

HINWEIS: Es kann etwas dauern, bitte nicht unruhig werden.

Anwendung (Pfad, Name, Assemblies)

Möchten Sie mehr über die aktuelle Anwendung erfahren, finden Sie die Informationen in verschiedenen Klassen:

```
Private Sub anwendungToolStripMenuItem_Click(sender As Object,
                e As EventArgs) Handles anwendungToolStripMenuItem.Click
```

```
        ListBox1.Items.Add("CommandLine: " & Environment.CommandLine.ToString())
        ListBox1.Items.Add("CurrentDirectory: " & Environment.CurrentDirectory)
        ListBox1.Items.Add("UserInteractive: " & Environment.UserInteractive.ToString())
        ListBox1.Items.Add("WorkingSet: " & Environment.WorkingSet)
        ListBox1.Items.Add("ProductName: " & Application.ProductName)
        ListBox1.Items.Add("ProductVersion: " & Application.ProductVersion)
        ListBox1.Items.Add("CompanyName: " & Application.CompanyName)
        ListBox1.Items.Add("ExecutablePath: " & Application.ExecutablePath)
        Dim info As New _
            ApplicationServices.AssemblyInfo(System.Reflection.Assembly.GetExecutingAssembly())
        ListBox1.Items.Add("Copyright: " & info.Copyright)
        ListBox1.Items.Add("Description: " & info.Description)
        ListBox1.Items.Add("LoadedAssemblies: ")
        For Each asm As System.Reflection.Assembly In info.LoadedAssemblies
            ListBox1.Items.Add("     Assembly: " & asm.FullName)
        Next
        ListBox1.Items.Add("Title: " & info.Title)
        ListBox1.Items.Add("Trademark: " & info.Trademark)
    End Sub
```

Eine mögliche Ausgabe:

```
CommandLine: "C:\Temp\___RezepteKochbuch\15 System\06 Infos\bin\Debug\Infos.vshost.exe"
CurrentDirectory: C:\Temp\___RezepteKochbuch\15 System\06 Infos\bin\Debug
UserInteractive: True
WorkingSet: 35938304
ProductName: Infos
ProductVersion: 1.0.0.0
CompanyName: Ingenieurbüro Gewinnus
ExecutablePath: C:\Temp\___RezepteKochbuch\15 System\06 Infos\bin\Debug\Infos.EXE
Copyright: Copyright © Ingenieurbüro Gewinnus 2012
Description:
LoadedAssemblies:
     Assembly: mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
     Assembly: Microsoft.VisualStudio.HostingProcess.Utilities, Version=11.0.0.0, Culture=neutral, PublicKeyToken=b03f5f
     Assembly: System.Windows.Forms, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
     Assembly: System.Drawing, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
     Assembly: System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
     Assembly: Microsoft.VisualStudio.HostingProcess.Utilities.Sync, Version=11.0.0.0, Culture=neutral, PublicKeyToken=
     Assembly: Microsoft.VisualStudio.Debugger.Runtime, Version=11.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50
     Assembly: vshost, Version=11.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
     Assembly: Microsoft.VisualBasic, Version=10.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
     Assembly: System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

HINWEIS: Weitere Detailinformationen über die aktuelle Assembly lassen sich mittels Reflection abrufen (Typen, Klassen etc.).

Soundkarte(n)

Wer für Stimmung am Computer sorgen möchte, wird sicher auch etwas mehr über die Soundkarten erfahren wollen:

```
    Private Sub soundkartenToolStripMenuItem_Click(sender As Object,
                    e As EventArgs) Handles soundkartenToolStripMenuItem.Click
        Dim Query As New SelectQuery("Win32_SoundDevice")
        Dim mos As New ManagementObjectSearcher(Query)
        For Each mo As ManagementObject In mos.Get()
```

```
            ListBox1.Items.Add("Name: " & mo("Name").ToString())
            ListBox1.Items.Add("Manufacturer: " & mo("Manufacturer").ToString())
        Next
    End Sub
```

Ausgabe:

```
Name: High Definition Audio-Gerät
Manufacturer: Microsoft
Name: High Definition Audio-Gerät
Manufacturer: Microsoft
Name: HD Pro Webcam C910
Manufacturer: Logitech
```

CLR-Version

Last, but not least sollten wir auch einen Blick auf die CLR bzw. deren Versionsnummer werfen, dies schon in Anbetracht möglicher Service Packs, Bugs etc.:

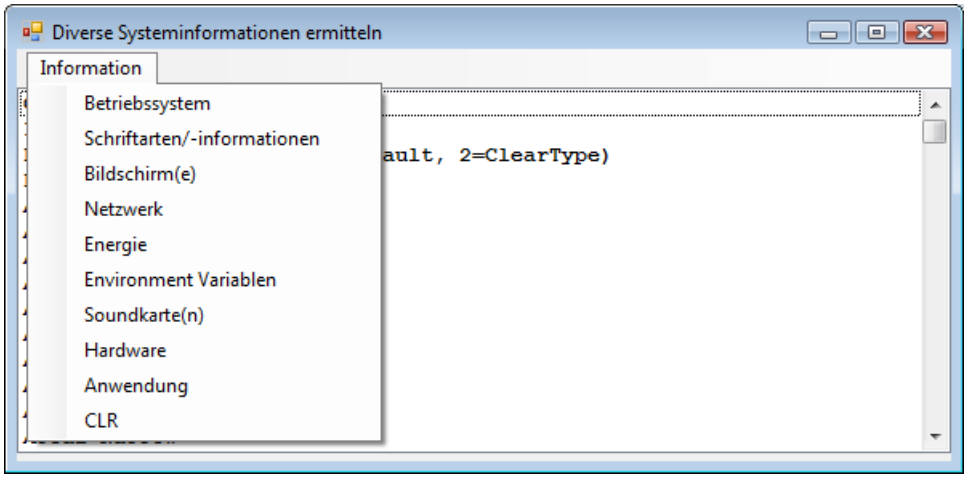
```
    Private Sub dToolStripMenuItem7_Click(sender As Object,
                        e As EventArgs) Handles dToolStripMenuItem7.Click

        ListBox1.Items.Add("CLR-Version: " & Environment.Version.ToString())
    End Sub
```

Mögliche Ausgabe:

```
CLR-Version: 4.0.30319.18010
```

Test

Starten Sie die Anwendung und wählen Sie über das Menü die gewünschte Information aus:



16.4.14 Sound per MCI aufnehmen

Im vorliegenden Beispiel wollen wir uns der Aufnahme von WAV-Dateien widmen, wobei (mittels unverwaltetem Code) auf Windows-Bordmittel, d h. auf die "gute alte" MCI-Schnittstelle, zurückgegriffen wird.

Oberfläche

Ein Formular mit einer Schaltfläche, die Sie mit *START* beschriften, genügt.

Quellcode

```
Imports System.IO
Imports System.Runtime.InteropServices

Public Class Form1
```

Die folgenden zwei API-Deklarationen greifen auf die MCI-Schnittstelle Ihres PCs zu:

```
    <DllImport("winmm.dll", CharSet:=CharSet.Auto)>
    Private Shared Function mciSendString(lpstrCommand As String,
                              lpstrReturnString As StringBuilder, uReturnLength As Integer,
                              hwndCallback As IntPtr) As Integer
    End Function

    <DllImport("winmm.dll", CharSet:=CharSet.Auto)>
    Private Shared Function mciGetErrorString(dwError As Integer,
                                lpstrBuffer As StringBuilder, uLength As Integer) As Integer
    End Function
```

MCI-Fehlermeldung dekodieren (wir erhalten ausführliche Fehlerbeschreibungen):

```
    Private Function getMciError(errCode As Integer) As String
        Dim errMsg As New StringBuilder(255)
        If mciGetErrorString(errCode, errMsg, errMsg.Capacity) = 0 Then
            Return "MCI-Fehler " & errCode
        Else
            Return errMsg.ToString()
        End If
    End Function
```

Mit Hilfe der API-Funktion *GetShortPathName* können wir den kurzen Pfadnamen zum Speicherort der WAV-Datei ermitteln, da ansonsten (z.B. wegen Leerzeichen im Dateinamen etc.) die WAV-Datei von *mciSendString* nicht gespeichert werden kann:

```
    <DllImport("kernel32.dll", CharSet:=CharSet.Auto)>
    Private Shared Function GetShortPathName(lpszLongPath As String,
                         lpszShortPath As StringBuilder, cchBuffer As Integer) As Integer
    End Function
```

```
    Private Function shortPathName() As String
        Dim shortPath As String = String.Empty
        Dim len As Long = 0
        Dim buffer As New StringBuilder(256)
        Dim s As String = Directory.GetCurrentDirectory()
        len = GetShortPathName(s, buffer, 256)
        shortPath = buffer.ToString()
        Return shortPath
    End Function
```

Nach all diesen Vorbereitungen kann es nun endlich losgehen. Bereits beim Laden des Formulars wird die MCI-Schnittstelle angesprochen:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Eine neue (leere) Sound-Aufnahme eröffnen:

```
        Dim mciString As String = "open new type waveaudio alias myAlias"
        Dim res As Integer = mciSendString(mciString, Nothing, 0, IntPtr.Zero)
        If res <> 0 Then
            MessageBox.Show(getMciError(res),
                            "MCI-Fehler beim Öffnen des Geräts ('Open New'-Befehl)",
                            MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
```

Parameter einstellen: 8000 Abtastungen pro Sekunde mit je 1 Byte (da es sich um die Standardwerte handelt, könnte dieser MCI-Aufruf auch weggelassen werden):

```
         mciString = "set myAlias time format ms bitspersample" &
                     " 8 channels 1 samplespersec 8000 bytespersec 8000"
        res = mciSendString(mciString, Nothing, 0, IntPtr.Zero)
        If res <> 0 Then
            MessageBox.Show(getMciError(res),
                 "MCI-Fehler beim Zuweisen der Parameter ('Set-Befehl)",
                 MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
    End Sub
```

Und schließlich der *START*-/*STOPP*-Button:

```
     Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        If Button1.Text = "START" Then                ' auf START geklickt
```

Die Aufnahme beginnt:

```
            Dim mciString As String = "record myAlias"
            Dim res As Integer = mciSendString(mciString, Nothing, 0, IntPtr.Zero)
            If res <> 0 Then
                MessageBox.Show(getMciError(res),
                      "MCI-Fehler beim Starten der Aufnahme ('Record-Befehl)",
                      MessageBoxButtons.OK, MessageBoxIcon.Error)
           End If
```

```
            Button1.Text = "STOPP"
        Else                                        ' auf STOPP geklickt
```

Die Sound-Datei wird im aktuellen Verzeichnis abgespeichert:

```
        Dim mciStr As String = "save myAlias " & shortPathName() & "\Test.wav"
            Dim res As Integer = mciSendString(mciStr, Nothing, 0, IntPtr.Zero)
            If res <> 0 Then
                MessageBox.Show(getMciError(res),
                        "MCI-Fehler beim Speichern der Datei ('Save'-Befehl)",
                        MessageBoxButtons.OK, MessageBoxIcon.Error)
                Button1.Text = "START"
            End If
        End If
    End Sub
End Class
```

Test

Vergewissern Sie sich, dass an Ihren PC ein Mikrofon angeschlossen ist und dass es korrekt funktioniert (siehe Bemerkungen).

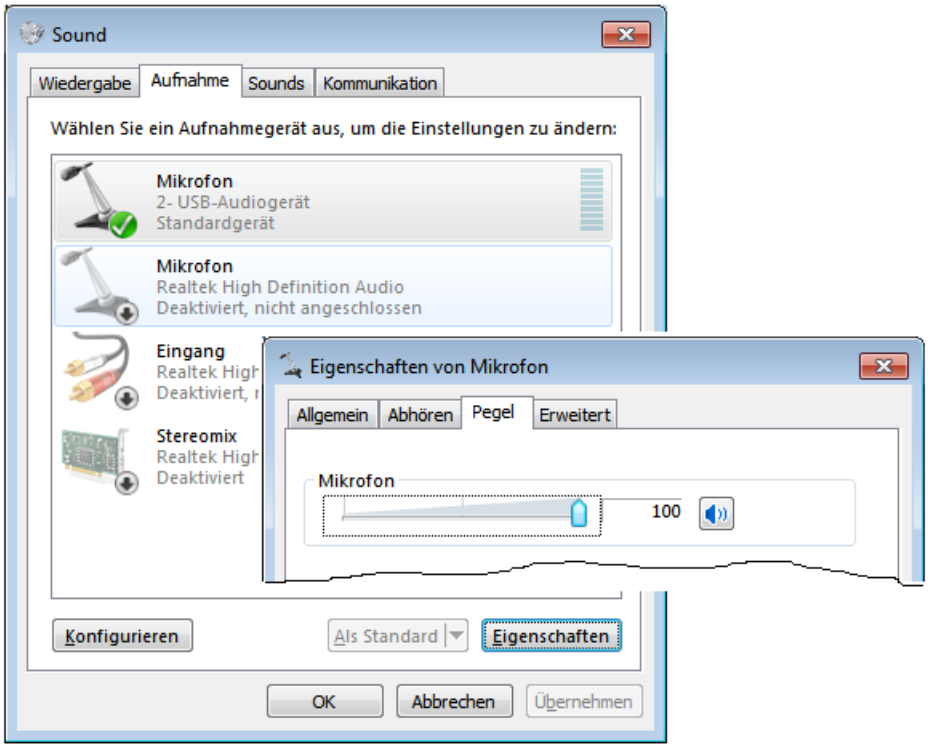
Nach Klick auf die *START*-Schaltfläche ändert sich deren Beschriftung in STOPP und die Aufnahme beginnt. Jetzt können Sie zum Beispiel eine Probe Ihrer Gesangskünste abliefern. Nach Aufnahmestopp findet sich die Sounddatei *Test.wav* im aktuellen Projektverzeichnis (bzw. im *.../bin/Debug*-Unterverzeichnis Ihres Projektordners).

Nach Doppelklick auf die Sounddatei dürfte sich normalerweise der Windows Media Player öffnen und Sie können sich das Ergebnis Ihrer gesanglichen Darbietungen anhören, oder aber Sie verwenden zur Wiedergabe das Beispiel

► 16.4.17 Sound-und Video-Dateien per MCI abspielen

Bemerkungen

- Über die Systemsteuerung (oder auch die Taskbar) haben Sie Zugriff auf die Dialoge für Mikrofonauswahl und Pegeleinstellungen:



- Bei schlechter Soundqualität sollten Sie vergleichsweise zunächst mit dem im Windows-Zubehör enthaltenen Sound Recorder einige Probeaufnahmen machen.
- Die ausführlichen MCI-Meldungstexte helfen Ihnen bei der Fehlersuche, zum Beispiel wenn ein falscher oder doppelt vergebener Alias verwendet wurde.

16.4.15 Mikrofonpegel anzeigen

Bei Soundaufnahmen wünscht man sich meist auch eine Anzeige des Mikrofonpegels, z.B. in Form eines Aussteuerungsbalkens. Im Folgenden wollen wir das Vorgängerbeispiel 16.4.14 mit einer einfachen Pegelanzeige ergänzen.

Oberfläche

Zum Formular des Vorgängerbeispiels fügen Sie eine *ProgressBar* (*Maximum* = 128) und einen *Timer* (*Interval* = 10) hinzu.

Quellcode

Ergänzen Sie zunächst den *Load*-Eventhandler des Formulars wie folgt:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        ...
```

Leider kann ein geöffnetes Audiogerät nicht gleichzeitig eine Sound-Datei aufnehmen und den Pegel anzeigen. Wir öffnen deshalb eine zweite Sound-Aufnahme mit unterschiedlichem Aliasnamen:

```
        Dim mciString As String = "open new type waveaudio alias myAlias"
        Dim res As Integer = mciSendString(mciString, Nothing, 0, IntPtr.Zero)
        If res <> 0 Then
            MessageBox.Show(getMciError(res),
                            "MCI-Fehler beim Öffnen des Geräts ('Open New'-Befehl)",
                            MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
```

Auf eine Parameterzuweisung können wir in unserem Fall verzichten, da mit den Standardwerten gearbeitet wird.

```
           Timer1.Start()
    End Sub
```

In der zweiten Programmergänzung wird im 10ms Takt der Pegel abgerufen:

```
    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
```

Der aktuelle Pegel kann mittels *status*-Befehl über das *level*-Flag abgefragt werden:

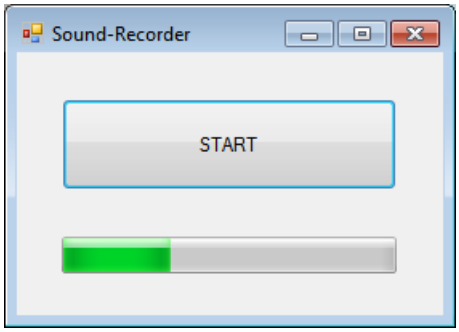
```
        Dim mciString As String = "status myAlias2 level"
        Dim buffer As New StringBuilder(20)
        Dim res As Integer = mciSendString(mciString, buffer, buffer.Capacity, IntPtr.Zero)
        If res <> 0 Then
            MessageBox.Show(getMciError(res),
                       "MCI-Fehler bei der Status-Abfrage  ('status ... level'-Befehl)",
                       MessageBoxButtons.OK, MessageBoxIcon.Error)
        End If
```

Der Input-Level ist ein vorzeichenloser Wert. Für die von uns gewählte 8-Bit Abtastgenauigkeit liegt er zwischen 0 und 127 (0x7F), bei 16-Bit Abtastung zwischen 0 und 32.767 (0x7FFF):

```
        Dim level As Byte = Byte.Parse(buffer.ToString())    ' 0 ... 127
        ProgressBar1.Value = level
    End Sub
End Class
```

Test

Die Aussteuerungsanzeige funktioniert sowohl vor als auch während der Aufnahme.



16.4.16 Pegeldiagramm aufzeichnen

So richtig professionell wirkt ein Sound Recorder erst dann, wenn man den Pegelverlauf in Echtzeit auf einem Diagramm verfolgen kann. Diese, u.a. auch bei medizinischen Überwachungsgeräten übliche Darstellung wollen wir anhand einer Programmerweiterung des Vorgängerbeispiels

► 16.4.15 Mikrofonpegel anzeigen

demonstrieren.

Oberfläche

Ergänzen Sie das vorhandene Formular durch eine *PictureBox* und ein repräsentatives *Label*. Gestalten Sie die Oberfläche entsprechend der Laufzeitabbildung am Schluss dieses Beispiels.

Quellcode-Ergänzungen

Um wegen der zahlreichen Ergänzungen die Übersicht nicht zu verlieren, empfiehlt es sich insbesondere bei diesem Beispiel, erst den kompletten Code von den Buchdaten in die Entwicklungsumgebung zu laden und dann mit der Lektüre fortzufahren.

```
...
Imports System.Drawing                      ' für die Grafikoperationen
```

Zunächst sind einige globale Variablen hinzuzufügen:

```
  Private g As Graphics = Nothing             ' Graphics-Objekt für PaintBox
    Private BA() As Byte = Nothing              ' Puffer-Array für die Pegelwerte
    Private xmax As Single                      ' Breite des Diagramms (Pixel)
    Private y0 As Single        ' y-Koordinate der Diagramm-Mittellinie (Pixel)
    Private v As Single         ' Streckungsfaktor der Amplitude
    Private i As Integer        ' Zähler für Timer-Ticks ab Start
    Private dStart As DateTime                  ' Startzeit der Aufnahme
    Private msec As Integer                     ' seit Start abgelaufene Zeit in Millisekunden

    Private iv As Integer                       ' Timer-Intervall (ms)
    Private Const tmax As Integer = 60000       ' Länge der Zeitachse (ms)
```

Zwecks Initialisierung der globalen Variablen wird der *Load*-Eventhandler ergänzt:

```
Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
    g = PictureBox1.CreateGraphics()
    iv = Timer1.Interval              ' Voreinstellung 10ms
```

Die erforderliche Größe des Puffer-Arrays ergibt sich aus obigen Konstanten:

```
    ReDim BA(tmax / iv)               ' 6000 gespeicherte Werte (ca. 60sek)
```

Abmessungen des Diagramms:

```
    xmax = PictureBox1.Width
    y0 = PictureBox1.Height / 2
```

Der vom MCI gelieferte Maximalpegel beträgt 127. Da bei Vollaussteuerung der obere bzw. untere Rand der *PictureBox* nicht ganz erreicht werden soll, bauen wir in den Streckungsfaktor eine kleine Reserve ein:

```
    v = y0 / 140
    ...
End Sub
```

Der *Click*-Eventhandler des Buttons wird modifiziert:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
      If Button1.Text = "START" Then          ' auf START geklickt
          Dim mciString As String = "record myAlias"
          Dim res As Integer = mciSendString(mciString, Nothing, 0, IntPtr.Zero)
          If res <> 0 Then
              MessageBox.Show(getMciError(res),
                    "MCI-Fehler beim Starten der Aufnahme ('Record-Befehl)",
                    MessageBoxButtons.OK, MessageBoxIcon.Error)
          End If
          Button1.Text = "STOPP"
```

Nach dem Klick auf START beginnt eine neue Aufnahme, d h., die Startzeit wird zugewiesen, der Zähler zurückgesetzt und das alte Diagramm gelöscht:

```
          dStart = DateTime.Now
          i = 0
          PictureBox1.Refresh()
      Else                                ' auf STOPP geklickt
```

Das Abspeichern der Sound-Datei wird in eine separate Prozedur (siehe unten) verlagert:

```
          saveFile()
          Button1.Text = "START"
      End If
  End Sub
```

Die Prozedur zum Abspeichern der Sound-Datei im aktuellen Verzeichnis:

```
  Private Sub saveFile()
      Dim mciStr As String = "save myAlias " & shortPathName() & "\Test.wav"
```

```
        Dim res As Integer = mciSendString(mciStr, Nothing, 0, IntPtr.Zero)
        If res <> 0 Then
            MessageBox.Show(getMciError(res),
                "MCI-Fehler beim Speichern der Datei ('Save'-Befehl)", MessageBoxButtons.OK,
                                                                        MessageBoxIcon.Error)
        End If
    End Sub
```

Bei einem Timerintervall von 10ms und 6000 Durchläufen ist die Endzeit in der Regel deutlich länger als die erwarteten 60 Sekunden[1]. Wollen Sie eine exakte Anzeige der seit Start abgelaufenen Sekunden hinzufügen, so kommen Sie um ein *TimeSpan*-Objekt nicht herum.

Der Aufbau der Grafik erfolgt synchron mit der Pegelabfrage. Der vorhandene *Tick*-Eventhandler des *Timer*s ist deshalb wie folgt zu ergänzen:

```
Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
    ...
```

Die folgende Zeile ist kein Fehler, denn während die Soundaufnahme läuft, hat der Button die Beschriftung STOPP:

```
    If Button1.Text = "STOPP" Then
        Dim ts As New TimeSpan(DateTime.Now.Ticks - dStart.Ticks)
        Dim sec As Integer = Convert.ToInt32(ts.TotalSeconds)
```

Anzeige der abgelaufenen Zeit (Sekunden):

```
        Label1.Text = sec.ToString()
```

Abbruchkriterium bei 60 Sekunden:

```
        If sec = tmax / 1000 Then
```

Speichern am Schluss:

```
            saveFile()
            Button1.Text = "START"
            Return
        End If
```

Um die x-Koordinate exakt berechnen zu können, muss die Auflösung zumindest im Millisekundenbereich liegen:

```
        msec = Convert.ToInt32(ts.TotalMilliseconds)
        Dim x As Single = msec * xmax / tmax
```

Pegel in Diagramm eintragen:

```
            g.DrawLine(New Pen(Color.Red, 0.1F), x, y0 - level * v, x, y0 + level * v)
```

Abspeichern des Pegels im Puffer-Array:

```
            BA(i) = level
```

[1] Windows ist kein Echtzeit-Betriebssystem und Prozesse mit höherer Priorität können solche mit niedrigerer Priorität ausbremsen.

```
                i += 1
        End If
    End Sub
```

Damit nach einem Verdecken des Formulars durch andere Fenster das Diagramm nicht auf Nimmerwiedersehen verschwindet, muss es im *Paint*-Eventhandler der *PaintBox* komplett neu gezeichnet werden:

```
    Private Sub PictureBox1_Paint(sender As Object, e As PaintEventArgs) _
                                                    Handles PictureBox1.Paint
        Dim g As Graphics = e.Graphics
```

Abstand zwischen zwei gespeicherten Werten auf Position *i*:

```
        Dim dx As Single = xmax / tmax * msec / i
```

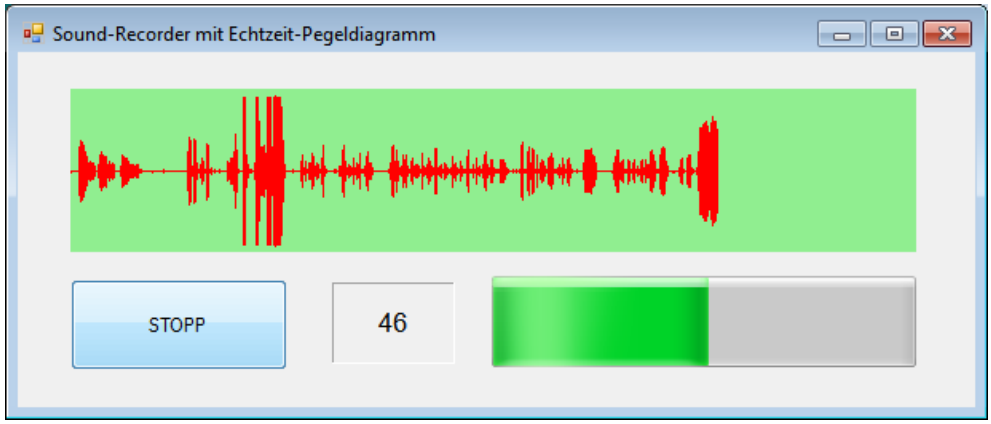
Der Pufferspeicher wird bis zur Position *i* ausgelesen und angezeigt:

```
        For j As Integer = 0 To i - 1
            g.DrawLine(New Pen(Color.Red, 0.1F), j * dx, y0 - BA(j) * v, j * dx, y0 + BA(j) * v)
        Next j
    End Sub

End Class
```

Test

Jetzt können Sie unmittelbar während der Sound-Aufnahme den kontinuierlichen Pegelverlauf "live" beobachten (siehe folgende Abbildung).

Bei STOPP, bzw. nach Ablauf von 60 Sekunden, wird die Datei *Test.wav* automatisch im Projektverzeichnis abgelegt.



16.4.17 Sound-und Video-Dateien per MCI abspielen

Für die Wiedergabe von Sound und Videos in Windows Forms-Anwendungen gibt es bereits einige vorgefertigte .NET- oder auch ActiveX-Steuerelemente (*SoundPlayer*, *WindowsMediaplayer*, ...).

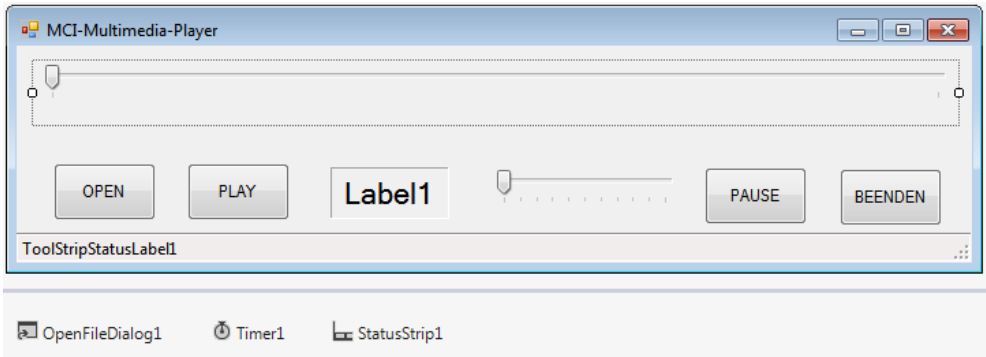
Wollen Sie aber individuelle Wünsche realisieren, wie zum Beispiel eine eigene Benutzeroberfläche, oder einfach nur mit verschiedenen Befehlen experimentieren, so können Sie die MCI-Schnittstelle direkt ansprechen. Diese allerdings ist nicht Teil von .NET, sondern gehört zum Uralt-Bestand von Windows (seit 3.1), d.h., die Fremdbibliothek *winmm.dll* muss importiert werden.

Mittels einfacher Kommandos (*Open*, *Play*, ...) können Sie dann auf alle verfügbaren Multimedia-Geräte zugreifen, für die ein MCI-Treiber existiert.

Das vorliegende Beispiel demonstriert dies anhand eines einfachen "selbstgestrickten" Players für Sound- und Videodateien unterschiedlichster Formate (Wave, MP3, Midi, AVI, WMV, MPEG etc.).

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie die Oberfläche von *Form1* mit vier *Button*s, zwei *TrackBar*s und je einem *Label*, *OpenFileDialog*, *Timer* und *StatusStrip*. Setzen Sie die *TopMost*-Eigenschaft von *Form1* auf *True*.



Klasse C_MCI

Den Zugriff auf die MCI-Schnittstelle kapseln wir zwecks späterer Wiederverwendbarkeit in einer separaten Klasse *C_MCI*, deren Gerüst wir über das Menü *Projekt/Klasse hinzufügen...* erzeugen:

```
...
Imports System.IO
Imports System.Text
```

Vergessen Sie keinesfalls, für den (unmanaged) Zugriff auf die Windows-API den folgenden Namespace einzubinden:

```
Imports System.Runtime.InteropServices

 Public class C_MCI
```

Die beiden erforderlichen MCI-Funktionen sind Teil der Datei *winmm.dll*. Dabei definiert der (optionale) *CharSet*-Parameter die automatische Übergabe von Zeichenfolgen (Marshalling) entsprechend des jeweiligen Zielbetriebssystems.

```
<DllImport("winmm.dll", CharSet:=CharSet.Auto)>
```

Die Funktion *mciSendString* ist das Herzstück des Programms, sie sendet unsere Befehle an die MCI-Schnittstelle:

```
Private Shared Function mciSendString(lpstrCommand As String,
                        lpstrReturnString As StringBuilder, uReturnLength As Integer,
                        hwndCallback As IntPtr) As Integer
End Function
```

Die MCI-Funktion *mciGetErrorString* gibt uns Fehlertexte für einen bestimmten Fehlercode zurück, was insbesondere beim Experimentieren sehr hilfreich ist:

```
<DllImport("winmm.dll", CharSet:=CharSet.Auto)>
Private Shared Function mciGetErrorString(dwError As Integer,
                          lpstrBuffer As StringBuilder, uLength As Integer) As Integer
End Function
```

Mit der Funktion *getMciError* kapseln wir den Aufruf obiger API-Funktion:

```
Private Function getMciError(errCode As Integer) As String
    Dim errMsg As New StringBuilder(255)
    If mciGetErrorString(errCode, errMsg, errMsg.Capacity) = 0 Then
        Return "MCI-Fehler " & errCode
    Else
        Return errMsg.ToString()
    End If
End Function
```

Ein bestimmtes MCI-Gerät wird innerhalb der Befehlszeichenkette über einen frei definierbaren Alias identifiziert:

```
Private _alias As String = "mciAlias"
```

Eine private Zustandsvariable zeigt an, ob das angesprochene MCI-Gerät geöffnet ist:

```
Private _isOpen As Boolean = False
```

Der Lesezugriff:

```
    Public ReadOnly Property IsOpen As Boolean
    Get
        Return _isOpen
    End Get
End Property
```

Alles beginnt mit dem Befehl zum Öffnen einer Multimedia-Datei:

```
Public Sub Open(filename As String)
```

Vorher schließen wir eine eventuell zuvor noch geöffnete Datei:

```
If _isOpen Then Me.Close()
```

Nun wird überprüft, ob die übergebene Multimedia-Datei tatsächlich existiert:

```
If Not File.Exists(filename) Then
   MessageBox.Show("Die Datei '" & filename & "' ist nicht vorhanden!", "Fehler",
                   MessageBoxButtons.OK, MessageBoxIcon.Error)
End If
```

Jetzt kommen wir zum wohl wichtigsten Parameter, dem MCI-Befehlsstring. Dieser besteht aus einzelnen Kommandos, die durch Leerzeichen voneinander getrennt sind.

Der komplette MCI-Befehlsstring zum Öffnen einer Datei besteht aus dem *Open*-Kommando, dem Dateinamen, dem Medientyp und dem Alias:

```
Dim mciStr As String = "open " & Chr(34) & filename & Chr(34) &
                                          " type mpegvideo alias " & _alias
```

Notwendige Erklärung zu obiger Zeile: Da auch der Dateipfad Leerzeichen enthalten kann, führt das häufig zu unergründbaren Fehlern. Wir müssen deshalb den Dateinamen in doppelte Anführungszeichen einschließen, also gewissermaßen "String im String" realisieren, am einfachsten geht das unter VB mit *Chr(34)*.

Nun kann das MCI-Gerät geöffnet werden:

```
Dim res As Integer = mciSendString(mciStr, Nothing, 0, IntPtr.Zero)
```

Zur Fehleranzeige verwenden wir hier einfachheitshalber ein Meldungsfenster, könnten aber stattdessen auch eine eigene Fehlerklasse definieren und eine Ausnahme werfen (*Throw MCI_Exception*):

```
If res <> 0 Then
   MessageBox.Show(getMciError(res),
                   "MCI-Fehler beim Öffnen des Geräts ('Open'-Befehl)",
                   MessageBoxButtons.OK, MessageBoxIcon.Error)
   Exit Sub
End If
```

HINWEIS: Eine Fehlerbehandlung wie die obige haben wir bei allen MCI-Aufrufen implementiert, im Folgenden aber aus Platz- und Übersichtlichkeitsgründen darauf verzichtet (siehe Buch-Beispieldaten).

Das Zeitformat für Positionsangaben auf Millisekunden festlegen:

```
mciStr = "set " & Me._alias & " time format ms"
res = mciSendString(mciStr, Nothing, 0, IntPtr.Zero)
```

Unsere Multimedia-Datei ist nun geöffnet:

```
    _isOpen = True
End Sub
```

Was nun, ist die Kapselung anderer wichtiger MCI-Kommandos als Eigenschaften oder Methoden.

Die folgende Eigenschaft liefert die Abspiellänge der Multimedia-Datei (in Millisekunden):

```
Public ReadOnly Property Length As Integer
    Get
        Dim buffer As New StringBuilder(255)
        Dim res As Integer = mciSendString("status " & _alias & " length", buffer,
                             buffer.Capacity, IntPtr.Zero)
        Return Convert.ToInt32(buffer.ToString())
   End Get
End Property
```

Die Multimedia-Datei zwischen zwei Positionen abspielen:

```
Public Sub Play(von As Integer, bis As Integer)
   Dim res As Integer = mciSendString("Play " & _alias & " From " & von & " To " &
                                                                  bis, Nothing, 0, IntPtr.Zero)
End Sub
```

Die Multimedia-Datei in voller Länge abspielen:

```
Public Sub Play()
    Play(0, Me.Length)
End Sub
```

Die folgende Eigenschaft liest/schreibt die aktuelle Abspielposition (Millisekunden):

```
Public Property Position As Integer
 Get
     Dim buffer As New StringBuilder(261)
     Dim res As Integer = mciSendString("status " & _alias & " position",
                          buffer, buffer.Capacity, IntPtr.Zero)
     Return Convert.ToInt32(buffer.ToString())
 End Get
 Set(value As Integer)
     Dim res As Integer = mciSendString("seek " & _alias & " to " & value,
                                                        Nothing, 0, IntPtr.Zero)
     res = mciSendString("play " & _alias, Nothing, 0, IntPtr.Zero)
 End Set
End Property
```

Eine Pause:

Den Abspielvorgang unterbrechen:

```
Public Sub Pause()
    Dim res As Integer = mciSendString("Pause " & _alias, Nothing, 0, IntPtr.Zero)

End Sub
```

Den Abspielvorgang fortsetzen:

```
    Public Sub Resume_()
        Dim res As Integer = mciSendString("Resume " & _alias, Nothing, 0, IntPtr.Zero)
    End Sub
```

Die Lautstärke festlegen:

```
    Public Sub volume(wert As Integer)
        Dim res As Integer = mciSendString("setaudio " & _alias & " volume to " & wert,
                                                                  Nothing, 0, IntPtr.Zero)
    End Sub
```

Den Abspielvorgang stoppen und auf den Anfang zurücksetzen:

```
   Public Sub Stop_()
        Dim res As Integer = mciSendString("Stop " & _alias, Nothing, 0, IntPtr.Zero)
    End Sub
```

Ein geöffnetes MCI-Gerät schließen:

```
     Public Sub Close()
        If _isOpen Then
            Dim res As Integer = mciSendString("Close " & _alias, Nothing, 0, IntPtr.Zero)
            _isOpen = False
        End If
    End Sub

End Class
```

Form1

Nun steht der Realisierung unseres Multimedia-Players nichts mehr im Weg, und wir können die eingangs entworfene Bedienoberfläche beispielsweise mit folgendem Code hinterlegen:

```
Public Class Form1
```

Die beim Laden des Formulars zugewiesenen Anfangswerte könnten Sie stattdessen auch direkt im Eigenschaftenfenster zuweisen:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        TrackBar1.Maximum = 10000
        TrackBar1.LargeChange = 1
        TrackBar2.Maximum = 1000
        TrackBar2.Value = 1000
        TrackBar2.LargeChange = 100
        Timer1.Interval = 100
   End Sub
```

Eine Instanz unseres Multimedia-Players erzeugen:

```
   Private myMCI As New C_MCI()
```

Korrekturfaktor für *TrackBar*-Anzeige:

```
Private kf As Single = 1
```

Eine Multimedia-Datei öffnen (OPEN-Schaltfläche):

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    If OpenFileDialog1.ShowDialog() = DialogResult.OK Then
        myMCI.Open(openFileDialog1.FileName)
```

Bei der Berechnung des Korrekturfaktors ist eine Typkonvertierung erforderlich, da sonst die Nachkommastellen abgeschnitten werden:

```
        kf = Convert.ToSingle(myMCI.Length) / TrackBar1.Maximum
        ToolStripStatusLabel1.Text = OpenFileDialog1.FileName
        Button2.Text = "PLAY"
    End If
End Sub
```

Die Schaltfläche PLAY/STOPP:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If Button2.Text = "PLAY" Then
        myMCI.Play()
        Timer1.Enabled = True
        Button2.Text = "STOPP"
    Else
        myMCI.Stop_()
        Timer1.Enabled = False
        TrackBar1.Value = 0
        Label1.Text = "00:00"
        Button2.Text = "PLAY"
    End If
End Sub
```

Die Schaltfläche PAUSE/WEITER:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    If Button3.Text = "PAUSE" Then
        myMCI.Pause()
        Button3.Text = "WEITER"
    Else
        Button3.Text = "PAUSE"
        myMCI.Resume_()
    End If
End Sub
```

Es folgt eine Hilfsfunktion für die Bereitstellung der laufenden Zeitanzeige in Minuten und Sekunden. Anhand der Reglerposition von *TrackBar1* werden zunächst die Sekunden ermittelt. Mittels Integer- und Modulo-Division werden dann die Minuten und restlichen Sekunden berechnet und durch einen Doppelpunkt getrennt:

```
Private Function getMinutes() As String
```

```
        Dim sekunden As Integer = Convert.ToInt32((TrackBar1.Value * kf / 1000))
        Return (sekunden / 60).ToString("00") & ":" & (sekunden Mod 60).ToString("00")
    End Function
```

Im *Tick*-Event des *Timer*s werden der Regler der *Trackbar* vorwärts bewegt und die Zeitanzeige aktualisiert:

```
    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Dim val As Integer = Convert.ToInt32(myMCI.Position / kf)
        If val < TrackBar1.Maximum Then
            TrackBar1.Value = val
            Label1.Text = getMinutes()
```

Das Ende unserer MM-Datei ist erreicht:

```
        Else
            myMCI.Stop_()
            Timer1.Enabled = False
            TrackBar1.Value = 0
            Label1.Text = "00:00"
            Button2.Text = "PLAY"
        End If
    End Sub
```

Wenn Sie am Regler der *TrackBar* ziehen, können Sie die Abspielposition verändern:

```
    Private Sub TrackBar1_Scroll(sender As Object, e As EventArgs) Handles TrackBar1.Scroll
        Try
            Me.myMCI.Position = Convert.ToInt32(TrackBar1.Value * kf)
        Catch ex As Exception
            MessageBox.Show(ex.Message, Application.ProductName,
                MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End Sub
```

Die Lautstärke lässt sich mit der kleinen *TrackBar* steuern:

```
    Private Sub TrackBar2_Scroll(sender As Object, e As EventArgs) Handles TrackBar2.Scroll
        myMCI.volume(TrackBar2.Value)
    End Sub
```

Die Schaltfläche zum Beenden:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        Me.Close()
    End Sub
```

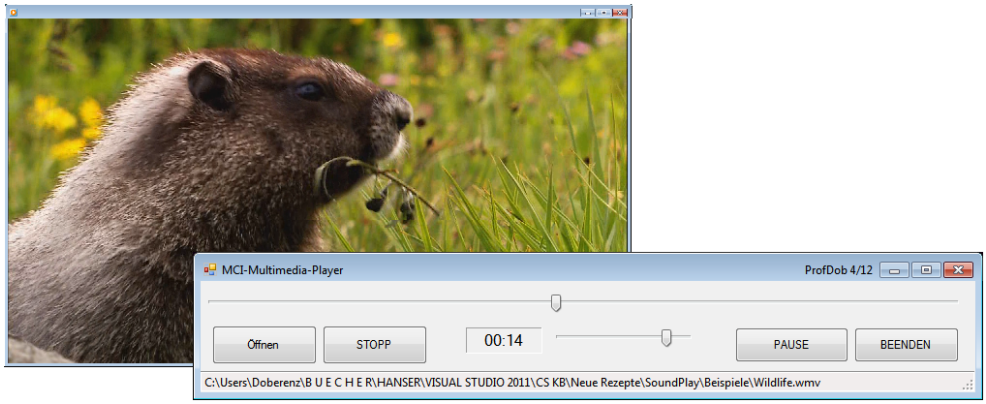
MCI-Player schließen:

```
    Private Sub Form1_FormClosing(sender As Object, e As FormClosingEventArgs) _
                                                                Handles MyBase.FormClosing
        myMCI.Close()
    End Sub
End Class
```

Test

Ring frei für Ihre Experimente! Starten Sie das Programm und laden Sie über die *Öffnen*-Schaltfläche eine beliebige Sound- oder Video-Datei (*.wav, .wma, .mp3, .midi, .avi, .wmv*). Spulen Sie mit dem oberen Regler vor und zurück und testen Sie die übrigen Bedienfunktionen.

Die Abbildung zeigt eine Momentaufnahme beim Abspielen der zu Windows mitgelieferten Videodatei *Wildlife.wmv*.



Bemerkungen

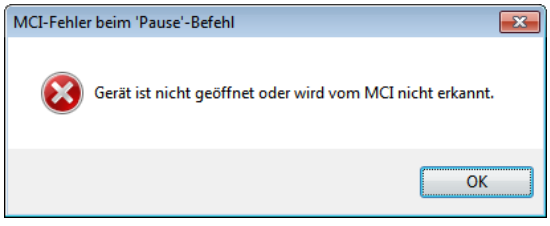
- In einer Windows Forms-Anwendung könnten Sie MCI auch durch die DirectX-Bibliothek ersetzen. Letztere müsste dann aber in vielen Fällen separat installiert werden, während hingegen jeder Computer von Haus aus auch über die erforderlichen MCI-Treiber verfügt.
- MCI bietet eine enorme Vielfalt von Befehlen für unterschiedlichste Dateiformate und Geräte, über die Sie sich am besten bei

  LINK: `http://msdn.microsoft.com/en-us/library/windows/desktop/dd743572(v=vs.85).aspx`

  informieren. So können Sie zum Beispiel mit dem folgenden Aufruf die Position und die Abmessungen des Fensters für die Video-Wiedergabe individuell festlegen:

```
mciSendString("put " & _alias + " window at " + x + " " + y + " " + width + " " +
              height, Nothing, 0, IntPtr.Zero)
```

- Auf fehlerhafte MCI-Kommandos werden Sie durch aussagekräftige Fehlermeldungen hingewiesen, zum Beispiel:



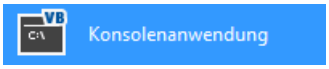
Kapitel 17

Konsolenanwendungen

Auch in Zeiten multimedialer, bunter und verspielter Windows-Oberflächen hat die gute alte Kommandozeilen- bzw. Konsolenanwendung, wie Sie sie bereits aus DOS-Urzeiten kennen, nach wie vor ihre Daseinsberechtigung. Sei es um ein kleines Hilfsprogramm zu schreiben, oder auch nur ein Beispielprogramm, bei dem es um Inhalte und nicht um optische Effekte geht.

17.1 Grundaufbau/Konzepte

Eine Konsolenanwendung erstellen Sie in Visual Studio über die gleichnamige Vorlage (Menü *Datei|Neu|Projekt...*).

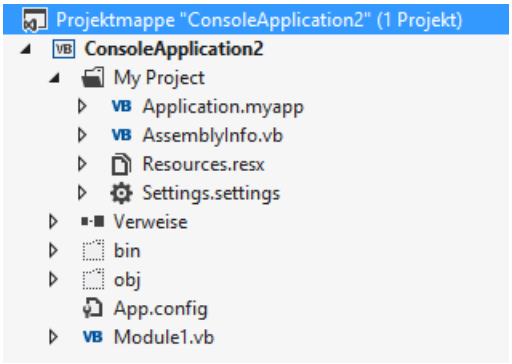


Wichtig ist in diesem Zusammenhang die Auswahl des gewünschten .NET-Zielframeworks, für das Sie die Konsolenanwendung entwickeln wollen.

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 VB-Quellcode von *Program.vb* 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.vb*), 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.

17.1.1 Unser Hauptprogramm – Module1.vb

Mit der schon geöffneten Datei *Module1.vb* liegt bereits das Hauptprogramm und damit der Einstiegspunkt vor Ihnen. Der VB-Code (Beispiel 17.1) enthält, im Gegensatz zu seinem C#-Pendant, lediglich den nackten Rumpf der *Sub Main*.

BEISPIEL 17.1: Unser Hauptprogramm

VB
```
Module Module1
    Sub Main()
    End Sub
End Module
```

Für C#-Umsteiger:

HINWEIS: In Visual Basic werden die Namespaces *Microsoft.VisualBasic*, *System*, *System.Collections*, *System.Collections.Generic*, *System.Data*, *System.Diagnostics* per Default importiert (*Projekt|Einstellungen|Verweise*) und tauchen deshalb auch nicht in der Liste der Importe auf.

Sub 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 jetzt um die "Nachrichtenschleife", d h. in diesem Fall die dauernde Abfrage der Eingabe, selbst kümmern. Bezogen auf unser erstes Beispiel bedeutet dies, dass die Anwendung sofort nach dem Start auch schon wieder beendet ist, da weder Daten abgefragt noch ausgegeben wurden.

BEISPIEL 17.2: Ein- und Ausgabe

VB
```
Module Module1
...
    Sub Main()
        Dim eingabe As String = ""
```

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()
    End Sub
...
```

HINWEIS: Rufen Sie obiges Beispiel direkt von der Kommandozeile aus auf, können Sie auf das *ReadKey* verzichten, da das Kommandozeilenfenster ja geöffnet bleibt.

17.1.2 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 *Main* eine Lösung. Neben der bereits vorgestellten Variante (*Sub*) ohne Rückgabewert können Sie diese Methode alternativ auch als *Function* mit einem *Integer*-Rückgabewert deklarieren. Dieser kann später z.B. in einer .BAT-Datei per %ERRORLEVEL% ausgelesen und verarbeitet werden.

BEISPIEL 17.3: Rückgabe und Auswertung eines *Integer*-Wertes

VB
```
Module Module1

    Function Main() As Integer
        Return 7
    End Function

End Module
```

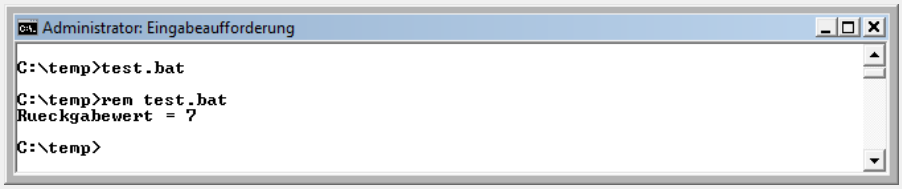
Die *Test.bat*:

```
rem test.bat
@echo off
test.exe
echo Rueckgabewert = %ERRORLEVEL%
```

BEISPIEL 17.3: Rückgabe und Auswertung eines *Integer*-Wertes

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.

17.1.3 Parameterübergabe

Kommandozeilenprogramme erhalten in vielen Fällen die nötigen Informationen als Parameter beim Programmstart. Wie Sie sicher schon vermutet haben, übernimmt ein String-Array-Parameter *args* der *Main*-Methode die Aufgabe, die übergebenen Parameter abzubilden.

BEISPIEL 17.4: Auswerten der Kommandozeilenparameter

VB

```
Module Module1
    Sub Main(args() As String)
        For Each arg As String In args
            Console.WriteLine(arg)
        Next
        Console.ReadKey()
    End Sub
End Module
```

Alternativ können Sie auch die *Environment.GetCommandLineArgs*-Collection auswerten, wie es folgendes Beispiel zeigt:

BEISPIEL 17.5: Auswerten der Kommandozeilenparameter per *GetCommandLineArgs*

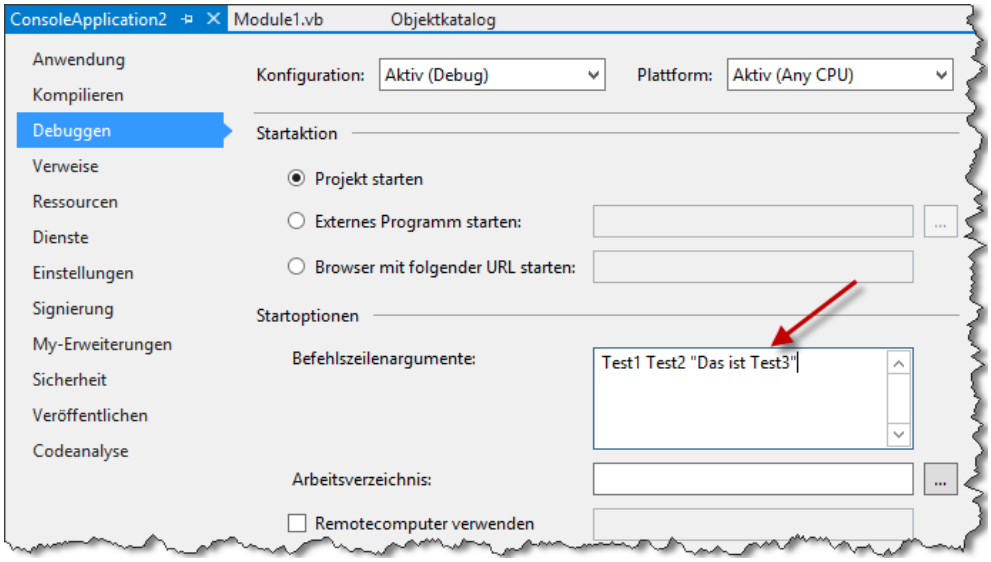
VB

```
Module Module1
    Sub Main()
        For Each arg As String In Environment.GetCommandLineArgs()
            Console.WriteLine(arg)
        Next
        Console.ReadKey()
    End Sub
End Module
```

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 folgenden Dialog auf und tragen Sie die Werte in die Textbox "Befehlszeilenargumente" ein.



17.1.4 Zugriff auf die Umgebungsvariablen

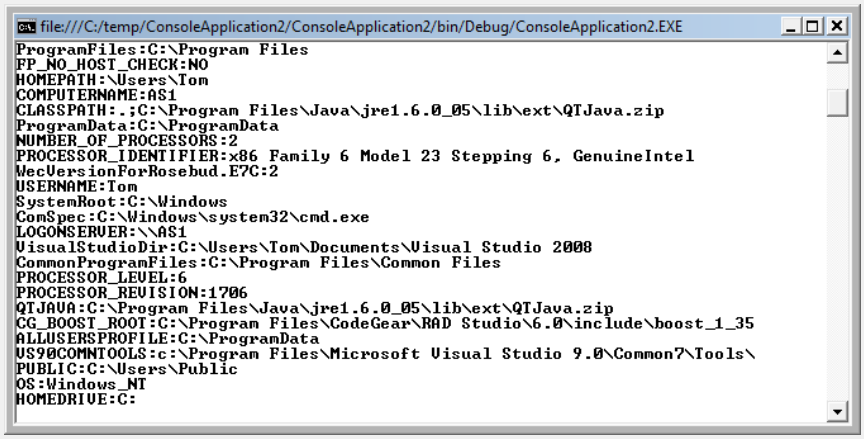
Was für den Windowsprogrammierer die Registrierdatenbank ist, sind für den Entwickler von Kommandozeilentools die Umgebungsvariablen. Über diese lässt sich recht bequem der Ablauf eines kleinen Programms steuern. Abrufen können Sie die Umgebungsvariablen über die *Environment.GetEnvironmentVariables*-Methode, diese gibt ein *Dictionary* zurück, das aus einer Kombination vom *Key* und *Value* besteht.

BEISPIEL 17.6: Abrufen aller Umgebungsvariablen

VB
```
...
    Sub Main()
        For Each de As System.Collections.DictionaryEntry In
                                Environment.GetEnvironmentVariables()
            Console.WriteLine("{0}:{1}", de.Key, de.Value)
        Next
        Console.ReadKey()
    End Sub
...
```

BEISPIEL 17.6: Abrufen aller Umgebungsvariablen

Ergebnis



Den direkten Zugriff auf eine einzelne Umgebungsvariable erhalten Sie über die Methode *GetEnvironmentVariable,* die den zugehörigen Wert zurückgibt:

SYNTAX: `Public Shared Function GetEnvironmentVariable (variable As String) As String`

BEISPIEL 17.7: Abfrage der PATH-Variablen

VB

```
...
    Sub Main()
        Console.WriteLine(Environment.GetEnvironmentVariable("PATH"))
        Console.ReadKey()
    End Sub
...
```

17.2 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.

17.2.1 Eigenschaften

Wichtige Eigenschaften von *Console* zeigt die folgende Tabelle:

Eigenschaft	Typ	Beschreibung
Title	*String*	Titel des Konsolenfensters
CursorVisible	*Boolean*	Sichtbarkeit des Kursors
CursorTop	*Integer*	Zeilenposition des Kursors
CursorLeft	*Integer*	Spaltenposition des Kursors
BackGroundColor	*ConsoleColor*	Hintergrundfarbe der Schrift
ForeGroundColor	*ConsoleColor*	Vordergrundfarbe der Schrift
BufferHeight, BufferWidth	*Integer*	Höhe und Breite des Pufferbereichs
CapsLock	*Boolean*	Ist die Feststelltaste gedrückt?
Error	*TextWriter*	Standard-Fehlerausgabestream
In	*TextReader*	Standard-Eingabestream
Out	*TextWriter*	Standard-Ausgabestream
WindowHeight, WindowWidth	*Integer*	die Fensterabmessungen
WindowTop,WindowLeft	*Integer*	Position des Fensterbereichs in Bezug auf den Bildschirmpuffer

17.2.2 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
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 Shared Event CancelKeyPress As ConsoleCancelEventHandler`

In den folgenden Abschnitten wollen wir an einigen kleinen Beispielen die Verwendung der obigen Member demonstrieren.

17.2.3 Textausgaben

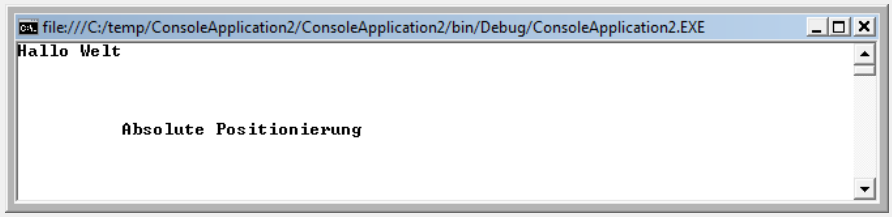
Wie der vorhergehenden Tabelle zu entnehmen ist, bieten sich für die Ausgabe von Text die Methoden *Write* und *WriteLine* an. Beide verfügen über eine beeindruckende Anzahl von Überladungen, die es ermöglichen, fast jeden relevanten Datentyp ohne vorherige Konvertierung zu übergeben. Beide Methoden 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 17.8: Verwendung *Write/WriteLine*

VB

```
Sub Main()
    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()
End Sub
```

Ergebnis



HINWEIS: Über die Eigenschaft *Console.Out.NewLine* haben Sie Einfluss auf die Art des Zeilenvorschubs, so können Sie den Wert beispielsweise auf "*vbCrLf + vbCrLf*" festlegen, um einen doppelten Zeilenvorschub zu erzielen.

17.2.4 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 17.9: Verwendung von Farbwerten

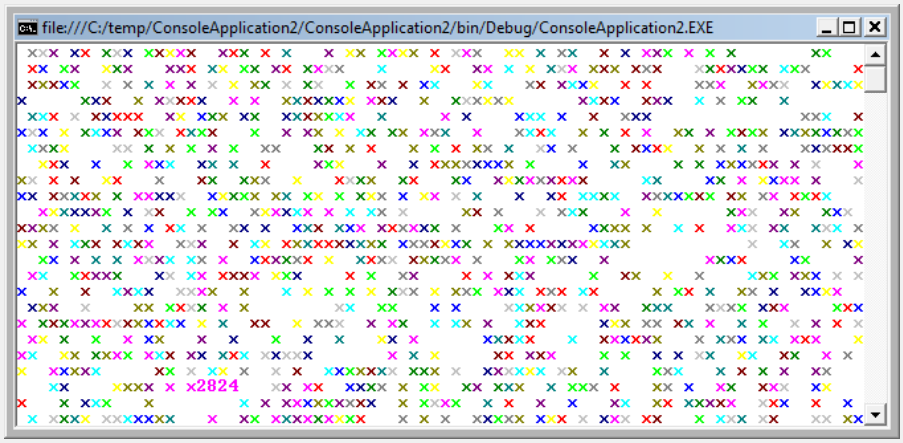
VB

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.

```
Sub Main()
    Const ym As Integer = 40
    Const xm As Integer = 80
    Dim xp As Integer = 0
    Dim yp As Integer = 0
    Console.CursorVisible = False
    Console.SetWindowSize(xm, ym)
    Dim rnd As New System.Random()
    Dim i As Integer = 0
    Do
        xp = Console.CursorLeft
        yp = Console.CursorTop
        Console.CursorLeft = rnd.Next(0, xm)
        Console.CursorTop = rnd.Next(0, ym)
        Console.ForegroundColor = CType(rnd.Next(1, 16), ConsoleColor)
        Console.Write("x")
        i += 1
    Loop While Not ((Console.CursorLeft = xp) AndAlso (Console.CursorTop = yp))
    Console.Beep(1000, 3000)
    Console.Write(i.ToString())
    Console.ReadKey()
End Sub
```

BEISPIEL 17.9: Verwendung von Farbwerten

Ergebnis



17.2.5 Tastaturabfragen

Die *ReadKey*-Methode liefert ein Objekt (genauer genommen eine Struktur) vom Typ *ConsoleKeyInfo*, dem man alle Informationen über die gedrückte Taste entnehmen kann.

Eigenschaften von *ConsoleKeyInfo:*

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 17.10: Verwendung von *ReadKey*

VB

Nach jeder Tastenbetätigung wird ein "Hallo" ausgegeben. Nur mittels *Esc*-Taste kann die Schleife verlassen werden.

```
Sub Main()
    Dim cki As ConsoleKeyInfo
    Do
        Console.WriteLine("Hallo")
        cki = Console.ReadKey(True)
    Loop While cki.Key <> ConsoleKey.Escape
End Sub
```

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 17.11: Es werden die gedrückte Tastenkombination und der zugehörige Unicode ausgegeben.

VB

```
Sub Main()
    Dim cki As ConsoleKeyInfo = Console.ReadKey(True)
    Dim sb As New System.Text.StringBuilder()
    sb.Append("Sie haben gedrückt die Taste(n): ")
    If cki.Modifiers <> 0 Then
        If (cki.Modifiers And ConsoleModifiers.Shift) <> 0 Then sb.Append("SHIFT+")
        If (cki.Modifiers And ConsoleModifiers.Alt) <> 0 Then sb.Append("ALT+")
        If (cki.Modifiers And ConsoleModifiers.Control) <> 0 Then sb.Append("CTL+")
    End If
    sb.Append(cki.Key.ToString() & "  (Zeichen: " + cki.KeyChar + ")")
    Console.WriteLine(sb.ToString)
End Sub
```

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*.

17.2.6 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 Visual Basic eine derartige Anwendung erstellen, die Eingaben wie oben gezeigt entgegennimmt, verarbeiten 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*.

Bei *Console.In* bzw. *Console.Out* handelt es sich um einen *TextReader* bzw. einen *TextWriter*, was die Verwendung wesentlich vereinfacht.

BEISPIEL 17.12: Verwendung von *Console.In* und *Console.Out*

VB

Einlesen des Eingabestreams und Ausgabe mit laufender Zeilennummer.

```
Sub Main()
    Dim zeile As String = ""
    Dim nr As Integer = 0
    zeile = Console.In.ReadLine()
```

Zeilenweises Verarbeiten des Eingabestreams solange Daten vorliegen:

```
    Do While zeile IsNot Nothing
```

Zeilenweise Ausgabe der gelesenen Daten:

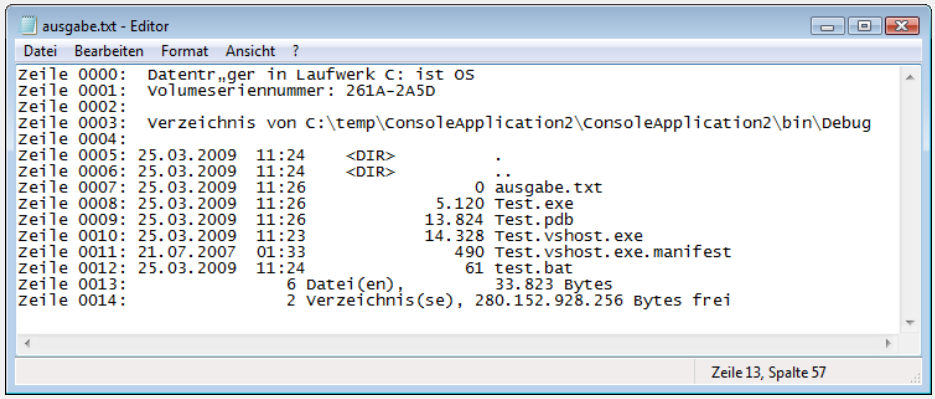
```
        Console.Out.WriteLine("Zeile {0}: {1}", nr.ToString("D4"), zeile)
        nr += 1
        zeile = Console.In.ReadLine()
    Loop
End Sub
```

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:



17.3 Praxisbeispiel: Farbige Konsolenanwendung

Das folgende, etwas längere, Beispiel soll Ihnen noch einmal die diversen Features der *Console*-Klasse demonstrieren.

```
Module Module1
```

Die folgende Methode zeichnet einen "Truck" in der Farbe *cc* am linken Rand der Zeile *y*:

```
    Private Sub drawTruck(cc As ConsoleColor, y As Integer)
```

Karosserie zeichnen:

```
Console.BackgroundColor = cc
Console.CursorTop = y
Console.CursorLeft = 5
Console.Write(" ")
Console.CursorLeft = 6
Console.Write(" ")
Console.CursorTop = y + 1
For i As Integer = 1 To 8
   Console.CursorLeft = i
   Console.Write(" ")
Next
```

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:

```
Sub Main()
    Const xm As Integer = 80   ' Fensterbreite
    Const ym As Integer = 40   '    "   höhe
    Const ypos As Integer = 5  ' Spur, in welcher Truck fährt
    Dim msg As String = "Drücken Sie eine beliebige Taste, um {0}"
```

Titel festlegen:

```
Console.Title = "Konsolen-Truck"
Console.SetWindowSize(xm, ym)
Console.WriteLine("Willkommen zur Truck-Demo!")
Console.Write("Welche Farbe soll der Truck haben? (rot=1, blau=2, grün=3) ")
```

Eingabe abfragen:

```
Dim farbe As String = Console.ReadLine()
```

Kursor ausblenden:

```
Console.CursorVisible = False
 Select Case farbe
    Case "1" : drawTruck(ConsoleColor.Red, ypos)
    Case "2" : drawTruck(ConsoleColor.Blue, ypos)
    Case "3" : drawTruck(ConsoleColor.Green, ypos)
    Case Else : drawTruck(ConsoleColor.White, ypos)
End Select
Console.SetCursorPosition(0, ypos + 4)
Console.ResetColor()
```

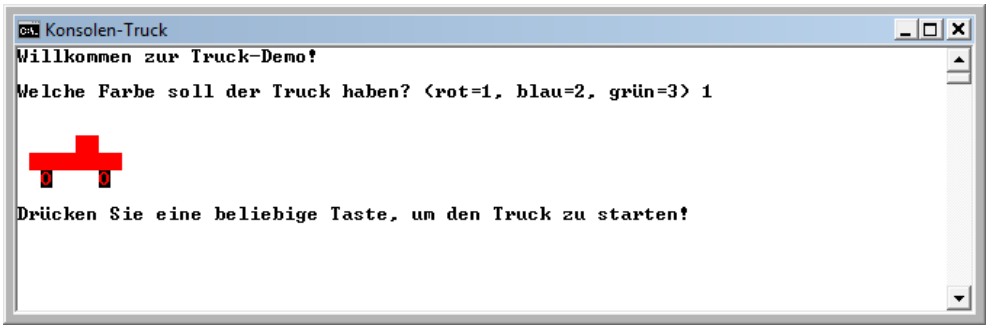
```
            Console.WriteLine(msg, "den Truck zu starten!")
            Console.ReadKey()
```

Soundausgabe:

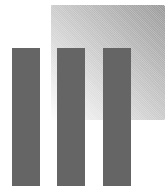
```
            Console.Beep()
```

Unser Truck "fährt" jetzt von links nach rechts:

```
            For i As Integer = 0 To xm - 10
               Console.MoveBufferArea(i, ypos, 9, 3, i + 1, ypos)
               System.Threading.Thread.Sleep(50)
            Next
            Console.Clear()
            Console.Beep()
            Console.SetCursorPosition(0, 0)
            Console.ResetColor()
            Console.WriteLine(msg, "die Truck-Demo zu beenden!")
            Console.ReadKey()
        End Sub
End Module
```



Damit wollen wir das Thema "Konsolenanwendungen" abschließen, denn das Haupteinsatzgebiet für Ihre Visual Basic-Anwendungen dürfte die Windows-Oberfläche sein.



Teil III: Windows Apps

- **Erste Schritte**
- **App-Oberflächen entwerfen**
- **Die wichtigsten Controls**
- **Apps im Detail**
- **App-Techniken**

Kapitel 18

Erste Schritte

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 VB-Programmierer in den nächsten Jahren zunehmend interessant werden könnten.

Ausgehend von den Grundlagen der Programmierschnittstelle *Windows Runtime* (kurz WinRT) zeigen wir Ihnen, wie Sie neben Ihren konventionellen Windowsanwendungen, basierend auf Windows Forms bzw. WPF, auch so genannte "Windows Apps" für Windows 8 bzw. Windows 10 entwickeln können[1]. Doch zunächst ein paar grundsätzliche Ausführungen, bevor wir uns mit den Feinheiten der Entwicklung auseinandersetzen werden.

HINWEIS: Nach derzeitigem Stand ist WinRT nur ab Windows 8 verfügbar, eine Portierung auf frühere Versionen, wie z.B. Windows 7 ist nicht angedacht. Damit dürfte für die nächste Zeit noch eine gewisse Einschränkung bei der Zielgruppe für Ihre Software gegeben sein.

18.1 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.

18.1.1 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 Windows Apps.

[1] Wir belassen es im weiteren beim Begriff "App".

Die Aufgaben als Laufzeitumgebung sind recht vielfältig:

- Schnittstelle für das Userinterface,
- Installation der Apps per Windows Store,
- Anzeige und Aktualisierung der Tiles[1] auf dem Desktop bzw. im Start-Menü (Windows 10),
- 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 14 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 924) 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 18.4 ab Seite 941.

18.1.2 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 **App**s" werden Anwendungen für Windows 8/8.1/10 bezeichnet, die auf der Windows Runtime basieren. Die folgende Abbildung zeigt Apps im Windows 10 Startmenü in verschiedenen Tiles-Größen:

[1] Die berühmt berüchtigten Kacheln ...

Diese Apps sind ebenfalls für die Darstellung und Verwendung 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.

HINWEIS: Diese Apps können, im Gegensatz zu den bekannten Desktop-Anwendungen, prinzipiell nur über den Windows-Store vertrieben bzw. installiert werden[1]. 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 VB-Programmierer interessant: Sie können Ihre XAML- und VB-Kenntnisse weiterverwenden[2], eine 1:1-Umsetzung von normalen WPF-Anwendungen ist jedoch meist nicht möglich (fehlende bzw. geänderte Namensräume, andere Controls, eingeschränkte Rechte).

18.1.3 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] Das ist eigentlich nicht ganz richtig, unter bestimmten Umständen ist auch ein sogenanntes Sideloading (siehe Abschnitt 22.5) möglich.

[2] 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 aktiv 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.

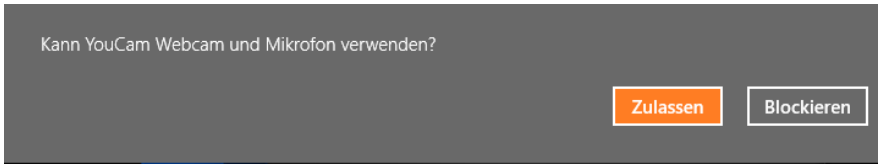
HINWEIS: Wer jetzt als Programmierer "der ersten Stunde" gequält an das mühsame Übergeben der Daten an den Vordergrundthread (Userinterface) denkt – vergessen Sie es! Alle Aktionen im Zusammenhang mit den beiden obigen Anweisungen laufen im Vordergrund-Thread ab, Sie können also direkt auf die Controls der Oberfläche zugreifen.

18.1.4 Process Sandboxing und Contracts

Mit der Umstellung auf eine Windows 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:

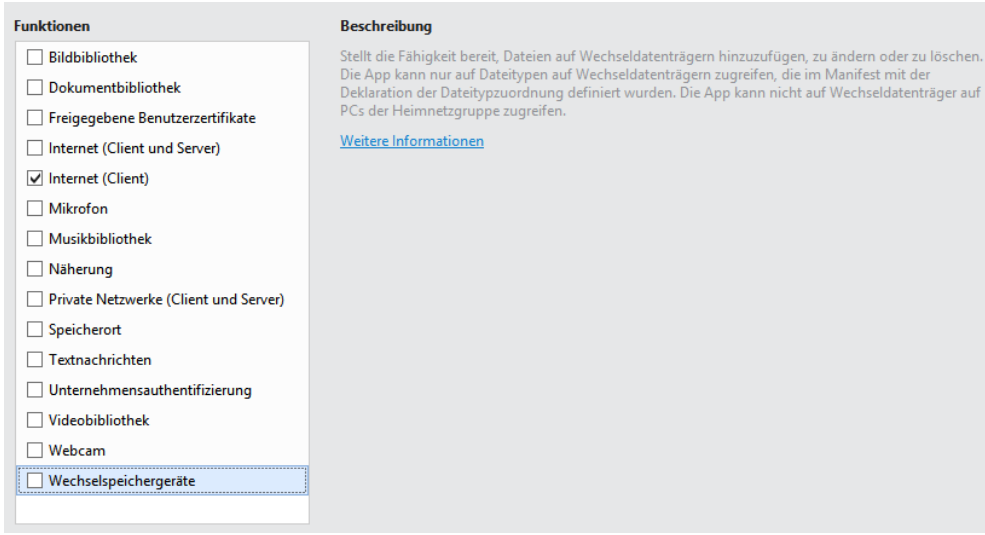
Genau an dieser Stelle kommen die so genannten *Contracts* ins Spiel. Jede App definiert, welche Schnittstellen nach außen benötigt werden. Im obigen Fall wird unter anderem der Zugriff auf die Webcam und das Mikrofon benötigt. Beim ersten Start der App werden Sie als Anwender von WinRT (nicht von der 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. 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 21, welche eingeschränkten Möglichkeiten zum Dateizugriff bestehen zeigt Ihnen Abschnitt 21.3 ab Seite 1058.

18.1.5 .NET WinRT-Profil

Für die Ausführung Ihrer VB-Apps ist neben der Windows Runtime (WinRT) zusätzlich eine Teilmenge des .NET Framework 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 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 8/10 nicht als App unter WinRT, dafür aber auf dem Desktop läuft.

18.1.6 Language Projection

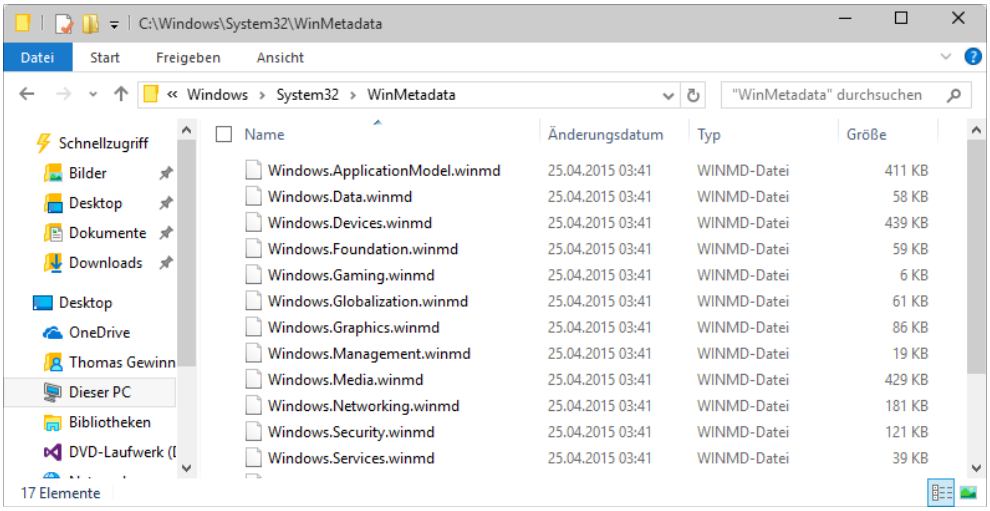
Mit *Language Projection* bezeichnet Microsoft eine Technik, die es ermöglicht, die eigentlichen COM-basierten WinRT-Objekte auf das Typsystem der jeweiligen Programmiersprache abzubilden, d h. zu projizieren. Dies ist auch der Grund, dass bei WinRT auf die Verwendung von InterOp-Assemblies verzichtet 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 VB-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 z.B. *HSTRING* auf *System.String,* wandelt die intern verwendeten *HRESULT*-Werte in VB-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.

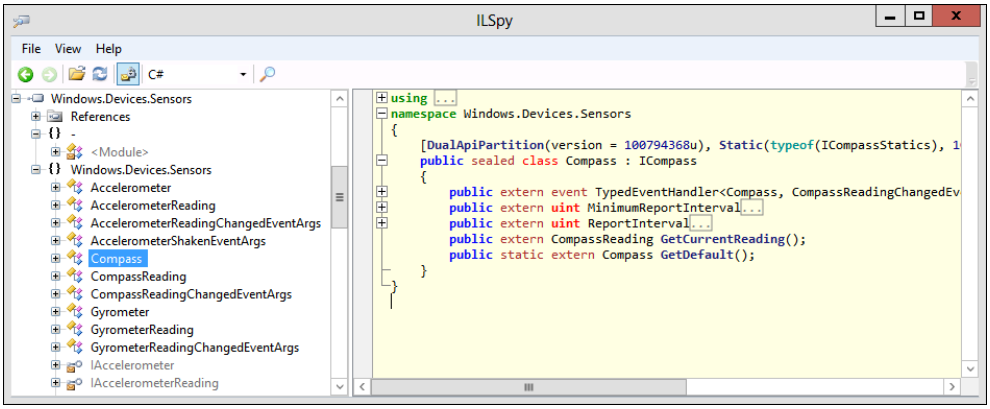
Als Entwickler müssen Sie sich also nicht an neue Bezeichner, Abläufe etc. gewöhnen, das nimmt ihnen alles der Compiler basierend auf den Metadaten der WinRT-Objekte ab.

[1] In Einzelfällen stehen Erweiterungsmethoden zur Verfügung, z.B. *AsStream* für die Umwandlung von *IInputStream* in *Stream*.

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/*

18.1.7 Vollbildmodus – war da was?

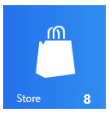
Mit der Vorstellung der ersten WinRT-Version bzw. mit Windows 8 hielt auch der sogenannte Vollbildmodus wieder Einzug auf dem Windows-PC. Statt eines Fensters wurde die App im Vollbild bzw. im optimalen Fall (ausreichende Bildschirmgröße) angedockt angezeigt.

Sie ahnen es sicher bereits, es hat etwas gedauert, aber mittlerweile hat auch Microsoft diese "Krankheit" überwunden. Apps laufen ab Windows 10 wie normale Desktop-Anwendungen im frei positionierbaren Fenster. Windows 8 bzw. 8.1-Nutzer sind jedoch nach wie vor an eine eingeschränkte Positionierbarkeit der Apps gebunden, die Größe der angedockten App kann jedoch mittlerweile frei gewählt werden.

18.1.8 Windows Store

Wer hätte das gedacht, nachdem eine bekannte Firma (die mit dem angebissenen Apfel) Wegbereiter dieses Trends war, will Microsoft natürlich auch nicht zurückstehen. Seit 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 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/`

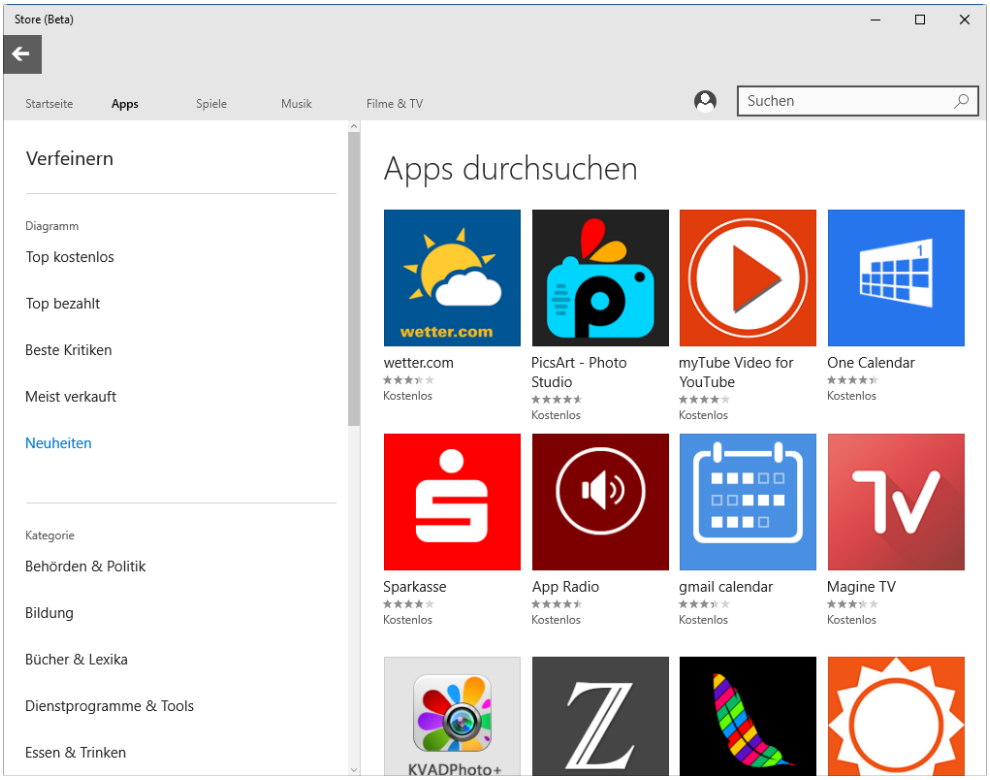
Nachfolgend steht Ihnen das komplette Angebot des Stores zur Verfügung. Neben Bewertungen und Verwendungsdetails wird Ihnen eine Vorschau sowie eine Beschreibung der App angezeigt. Mit dem Klick auf die Schaltfläche *Installieren* wird der Installationsvorgang angestoßen, die Fertigstellung wird durch eine kleine Textnachricht auf dem 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 14 Euro für eine Einzelperson bzw. 100 Euro für Unternehmen).
- 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).

[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 ...

- Apps werden im Allgemeinen als Lizenz zum Betrieb auf **mehreren** 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`

18.1.9 Zielplattformen

Hier wird es langsam aber sicher immer unübersichtlicher. Grundsätzlich müssen Sie zunächst klären, welche Zielplattform Ihnen "vorschwebt", dann entscheiden Sie sich für einen Projekttyp. Was sich so einfach anhört, zieht aber auch Abhängigkeiten nach sich.

Die Varianten nach Visual Studio-Projekttyp im Kurzüberblick:

Projekttyp	Bemerkung
Windows 8/Universal	Zielplattformen sind Windows 8 PCs und Windows Phone-Geräte. Das Projekt teilt sich intern für beide Zielplattformen auf. Die App läuft auch auf Windows 10, aber nicht auf früheren Versionen (< Windows 8).

Projekttyp	Bemerkung
Windows 8/Windows	Die App ist ausschließlich für den Windows 8-Desktop bestimmt und läuft mit Windows 8 (Intel) bzw. Windows RT (ARM). Die App läuft auch auf Windows 10, aber nicht auf früheren Versionen (< Windows 8).
Windows 8/Phone	Eine reine Windows Phone 8-App.
Windows Universal (Windows Apps[1])	Hierbei handelt es sich um die neue "eierlegende Wollmilchsau", die geräteübergreifend unter Windows 10 läuft, d h. Ihre App ist auf PC, Xbox, Phone etc. lauffähig. Selbstverständlich sind Anpassungen bei einer so breiten Hardwarebasis nötig, jedes System hat andere Schnittstellen, Geräte, Bildschirmgrößen etc. Der Nachteil dieses App-Typs ist die Beschränkung auf Windows 10, was noch für einige Jahre die Installationsbasis einschränkt, auch wenn die derzeitigen Windows 8-Nutzer wohl recht schnell auf Windows 10 wechseln werden.

HINWEIS: 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 ausschließlich auf Windows 8/10 als Zielplattform beschränken.

18.2 Entwurfsumgebung

Haben Sie sich für die Entwicklung von Apps entschieden, sollten Sie auch einen Blick auf die verfügbaren Tools bzw. auf die Grundvoraussetzungen der App-Entwicklung werfen.

18.2.1 Betriebssystem

Gleich ein wichtiger Hinweis vorweg:

HINWEIS: Apps können Sie je nach Typ nur unter Windows 8 bzw. Windows 10 entwickeln, d.h., Sie brauchen in jedem Fall einen passenden PC.

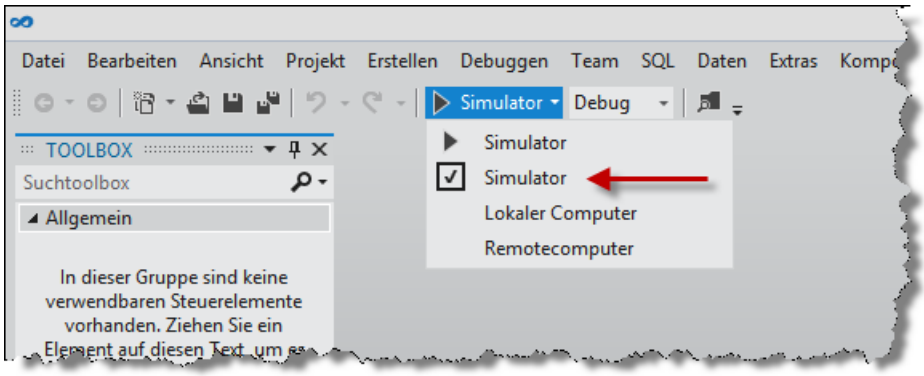
18.2.2 Windows-Simulator

Tablet-Entwicklung hin oder her, aber wollen Sie gleich ein oder mehrere Geräte für 500-1000 Euro kaufen, nur um ein paar Mal eine App unter den Bedingungen eines Tablets zu testen? Sicher nicht, es sei denn, Sie wissen schon vorher, dass Ihre App der absolute Renner wird.

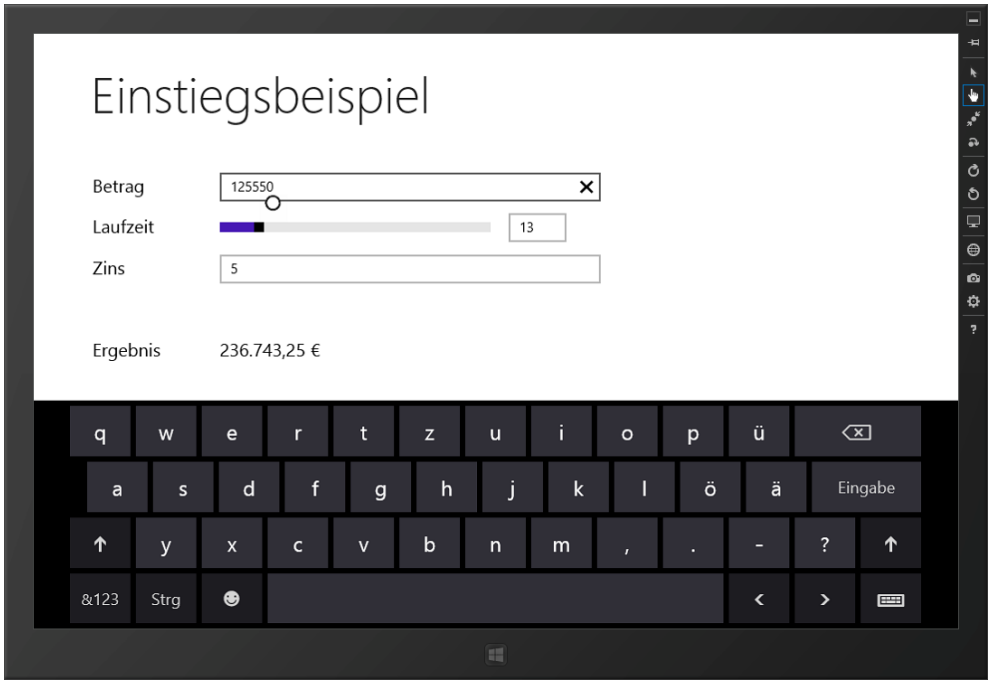
Hier hilft Ihnen der Windows-Simulator weiter.

[1] Das ist zum derzeitigen Stand die Bezeichnung.

Aktivieren Sie diesen über die Symbolleiste:

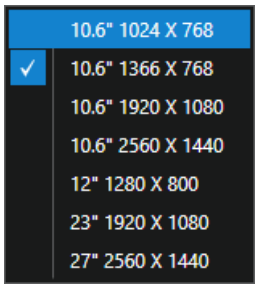


Starten Sie jetzt ein Projekt, wird dieses nicht auf dem normalen Windows 8-Desktop, sondern im Simulator wiedergegeben:

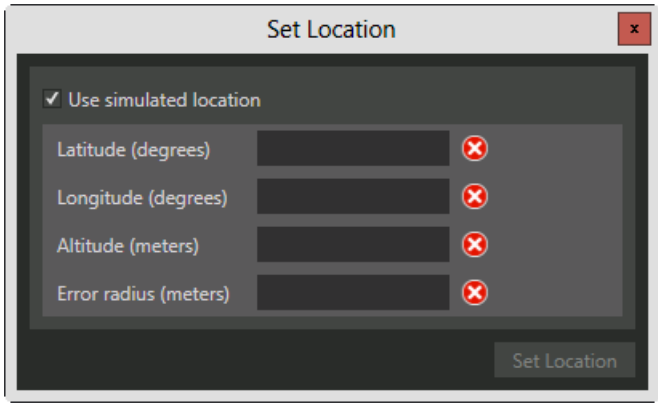


Dies hat gleich mehrere Vorteile:

- 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):



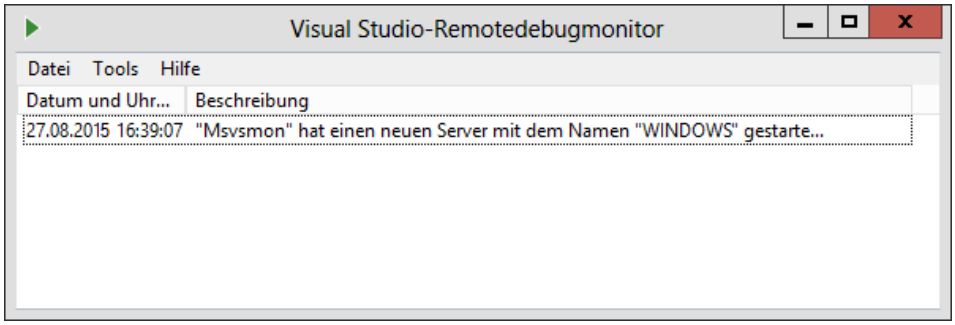
- Sie können Screenshots des aktuellen Inhalts anfertigen und diese in die Zwischenablage oder als Datei speichern lassen.

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.

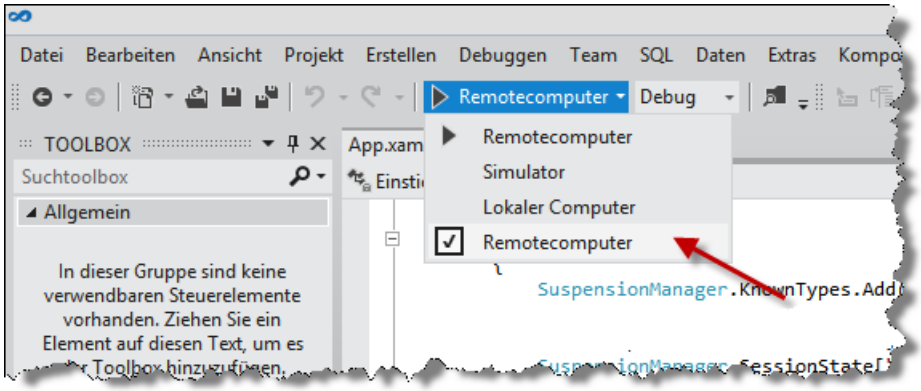
18.2.3 Remote-Debugging

Ein zweites hilfreiches Werkzeug für den WinRT-Entwickler ist der Remotedebugger. Installieren Sie diesen auf einem weiteren Windows 8/10-System (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.

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.

18.3 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 wollen wir uns hier nicht komplett wiederholen, wir verweisen statt dessen auf die PDF mit den entsprechenden WPF-Kapiteln.

18.3.1 Aufgabenstellung

Wir wollen eine einfache App zur Zins-Berechnung entwickeln. Dazu benötigen wir lediglich drei Eingabefelder, d.h. für

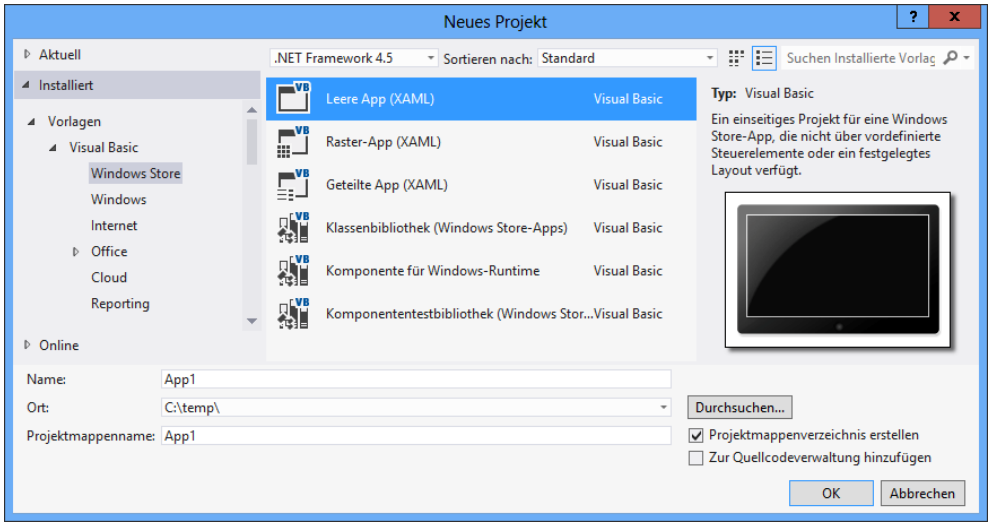
- den Anlagebetrag,
- den Zins
- und die Laufzeit.

Einfachheitshalber gehen wir von einer jährlichen Zahlweise aus. Ergebnis unserer Bemühungen ist der Endbetrag nach obiger Laufzeit.

18.3.2 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/Windows Universal* 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 bereits ein kleines Framework vorhanden, das sich um die wichtigsten Aufgaben im Lebenszyklus der 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 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:

```
Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Linq
Imports System.Text
Imports System.Threading.Tasks
```

[1] ... durch WinRT (Speichermangel) oder den Anwender (Wischbewegung von oben)

```
Namespace Einstieg
    Public Class Finanz
        Implements INotifyPropertyChanged
```

Die vier Eigenschaften:

```
        Private m_betrag As Double = 0

        Public Property Betrag() As Double
            Get
                Return m_betrag
            End Get
            Set(value As Double)
                m_betrag = value
                Calc()
            End Set
        End Property

        Private m_zins As Double = 0

        Public Property Zins() As Double
            Get
                Return m_zins
            End Get
            Set(value As Double)
                m_zins = value
                Calc()
            End Set
        End Property

        Private m_laufzeit As Integer = 0

        Public Property Laufzeit() As Integer
            Get
                Return m_laufzeit
            End Get
            Set(value As Integer)
                m_laufzeit = value
                Calc()
            End Set
        End Property
        Private m_endwert As Double = 0
```

Der Endwert ist aus naheliegenden Gründen schreibgeschützt:

```
        Public ReadOnly Property Endwert() As Double
            Get
                Return m_endwert
            End Get
        End Property
```

Eine zweite Variante dieser Eigenschaft liefert uns gleich einen formatierten *String* (Währungswert) statt des *Double*-Wertes:

```
        Public ReadOnly Property EndwertAsString() As String
            Get
                Return m_endwert.ToString("c")
            End Get
        End Property
```

Die Berechnung des Endwertes:

```
        Private Sub Calc()
            Try
                m_endwert = m_betrag * Math.Pow(1 + m_zins / 100, m_laufzeit)
            Catch generatedExceptionName As Exception
                m_endwert = 0
            End Try
```

Hier lösen wir auch das *PropertyChanged*-Ereignis aus:

```
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Endwert"))
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("EndwertAsString"))
        End Sub
    End Class
End Namespace
```

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.

18.3.3 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.vb*).

HINWEIS: Die erste sichtbare Page legen Sie in der Datei *App.xaml.vb* 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">
```

Die Zeilen:

```
<Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
</Grid.RowDefinitions>
```

Die Spalten:

```
<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.vb* und nehmen Sie folgende Erweiterung vor:

```
...
NotInheritable Class App
    Inherits Application

    Public Shared finanzrechner As New Einstieg.Finanz()
...
```

HINWEIS: Den Grund für die Definition in der *App*-Klasse erfahren Sie in den folgenden Abschnitten, deshalb bitte etwas Geduld.

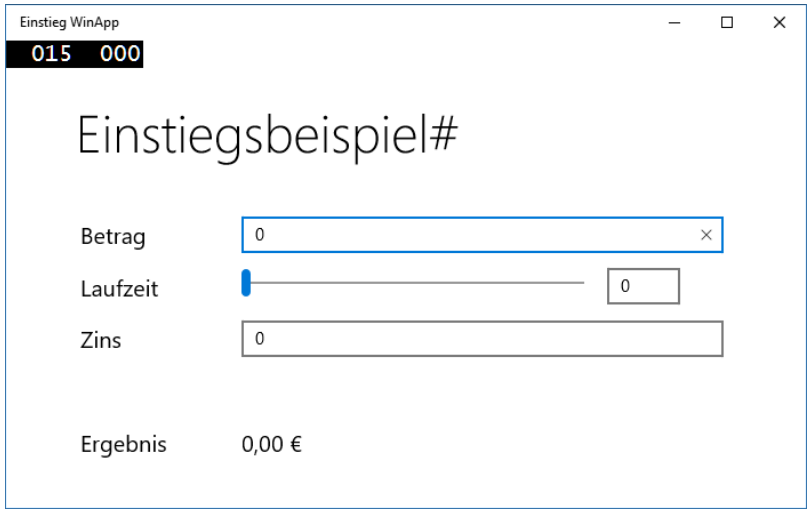
Die eigentliche Datenbindung realisieren wir in der Datei *MainPage.xaml.vb*:

```
Public NotInheritable Class MainPage
    Inherits Page

    Protected Overrides Sub OnNavigatedTo(e As Navigation.NavigationEventArgs)
        Me.DataContext = App.finanzrechner
    End Sub

End Class
```

18.3.4 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:

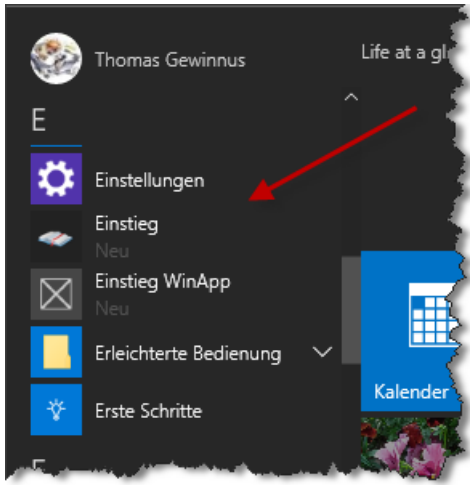
[1] In der vorliegenden Version war noch keine Stringformatierung wie in WPF implementiert.



So weit – so gut, das gleiche Programm hätten Sie mit minimalen Änderungen auch als WPF-Anwendung realisieren können.

HINWEIS: Die beiden Zählerfelder in der linken oberen Ecke sind die Anzeige der Frames. Sie können deren Anzeige über *App.xaml.vb*, Eintrag *EnableFrameRateCounter=False* deaktivieren

Doch lassen Sie sich nicht täuschen, Visual Studio hat die Anwendung nicht nur kompiliert, sondern ganz nebenbei auch gleich mit einer Entwicklerlizenz installiert (dies ist der Windows Runtime geschuldet):



18.3.5 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 *BlankPage.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 Property Laufzeit() As Integer
            Get
                Return m_laufzeit
            End Get
            Set(value As Integer)
                m_laufzeit = value
                Calc()
                RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Laufzeit"))
            End Set
        End Property
```

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.

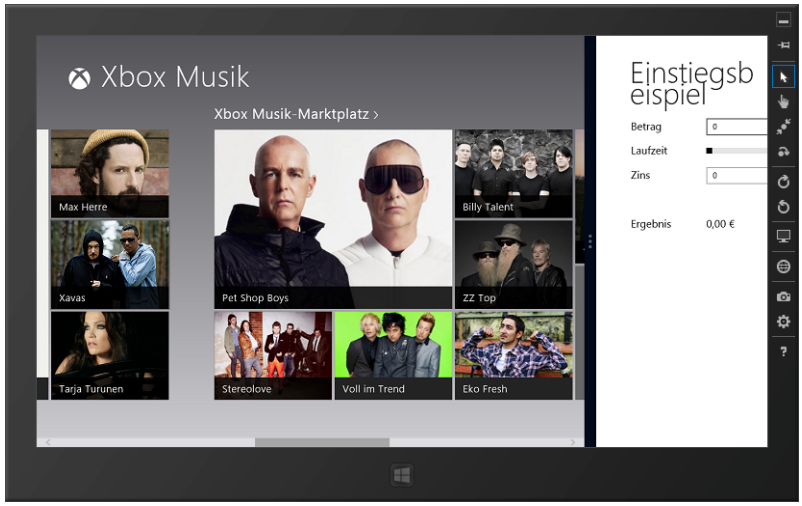
18.3.6 Fazit

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>
```

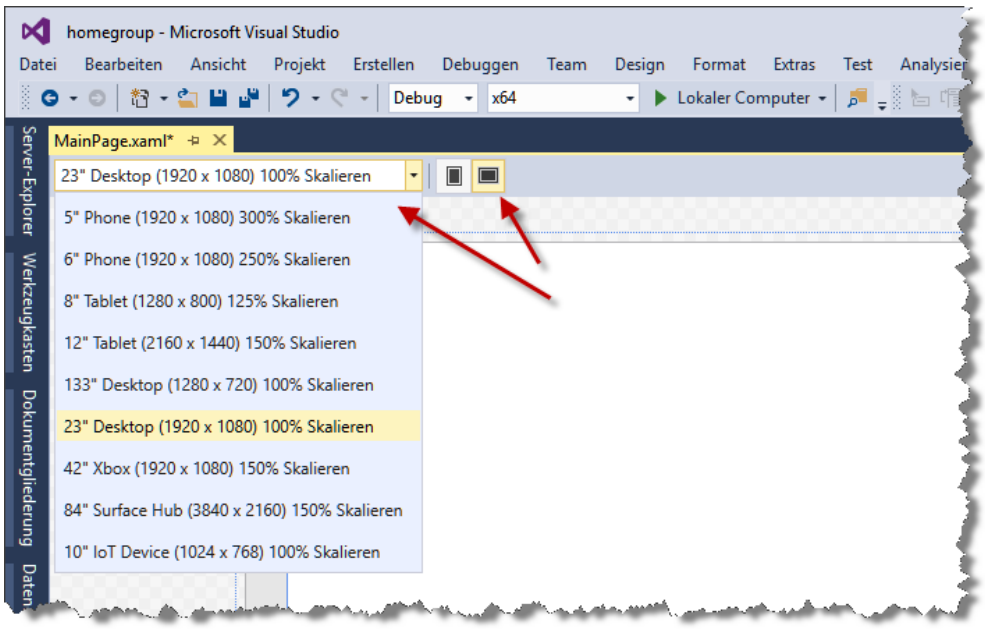
Prüfstein für Ihre Anwendung sollte in jedem Fall der Simulator in der niedrigsten Auflösung im Hochkant-Format sein. In dieser Konstellation steht am wenigsten Platz in horizontaler Richtung zur Verfügung (siehe folgende Abbildung).

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, wir müssten explizit auf diesen Modus reagieren und ein geändertes Layout anbieten. Doch dies soll nicht Gegenstand dieses Einführungsbeispiels sein.

So weit so gut, doch spezifisch für die neuen Windows 10 Apps ist die Möglichkeit, diese auch auf anderen Geräteklassen zum Laufen zu bringen. Entsprechende Emulatoren unterstützen Sie hier beim Test der App. Gleichzeitig können Sie aber auch schon im Entwurfsfenster Rücksicht auf die zukünftigen Geräte nehmen, entsprechende Profile werden "ab Werk" bereitgestellt:



HINWEIS: Sie sollten wenigstens die wichtigsten Auflösungen testen, bevor Sie die Kunden mit Ihren Apps beglücken.

18.4 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 ein Framework zur Installation und Verwaltung der Apps ist.

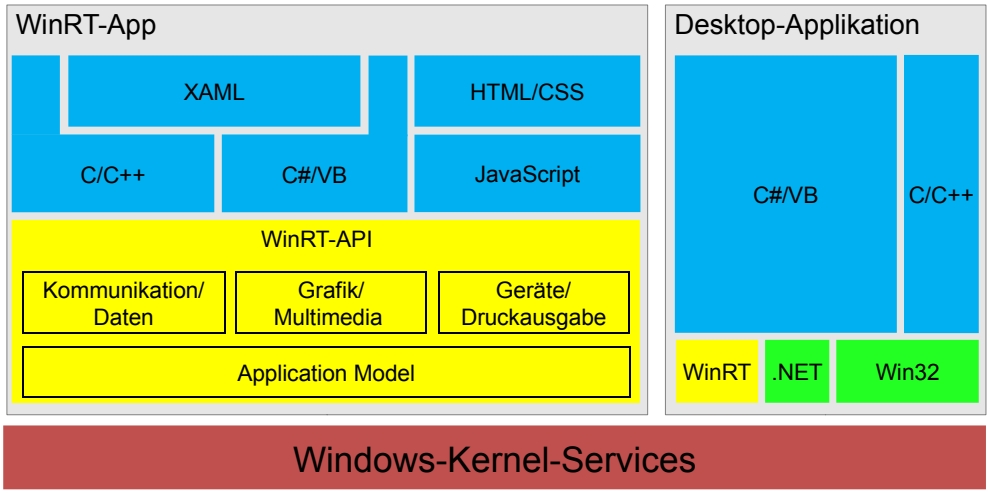
Doch welche Rolle spielt WinRT für Sie als VB-Programmierer?

18.4.1 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 Apps die unbedingte 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

► 18.5.1 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" Windows Apps. Wie Sie obiger Abbildung entnehmen können, lassen sich derzeit zwei Hauptentwicklungsrichtungen erkennen:

- XAML für den Oberflächenentwurf und C++, VB oder C# 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).

HINWEIS: Beachten Sie auch, dass bei 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).

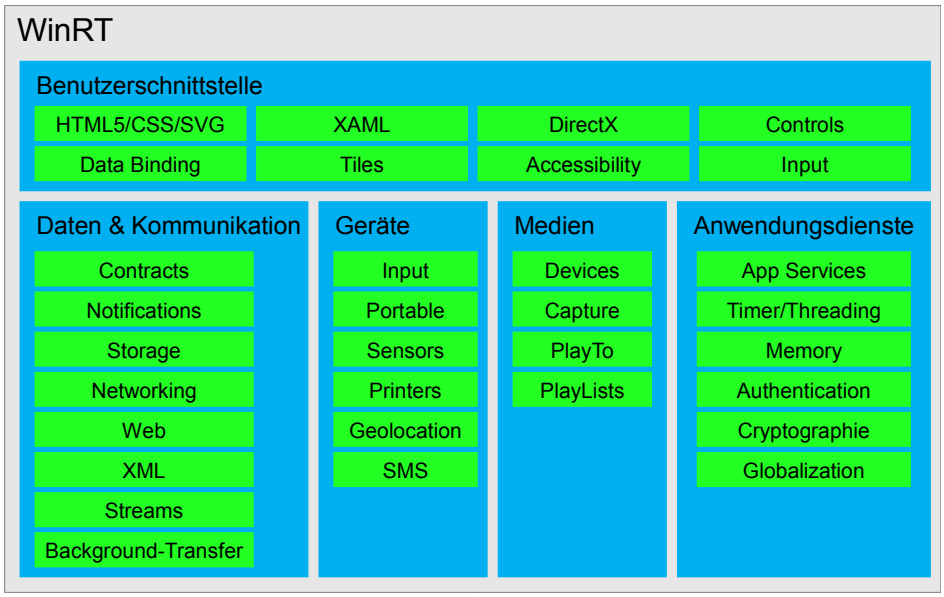
18.4.2 Die WinRT-API

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.

Benutzerschnittstelle

Hierbei handelt es sich um alle Aufgaben, die im Zusammenhang mit der Darstellung der Apps anfallen. Dies sind natürlich vor allem die Schnittstellen zu den WinRT-Controls, die Sie als VB-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. 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.)[2].

---

[1] Die HTML5+SVG-Unterstützung ist vor allem für die JavaScript-Programmierer von Interesse.

[2] 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.

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.

18.4.3 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

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 VB 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.VisualBasic
```

HINWEIS: Welche Unter-Namespaces im Einzelnen verfügbar sind, ist schnell ersichtlich, wenn Sie ein App-Projekt bearbeiten. Hier werden automatisch alle nicht verfügbaren Namespaces und deren Klassen ausgeblendet.

18.4.4 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 VB-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 VB-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 18.1.6) 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 VB-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

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:

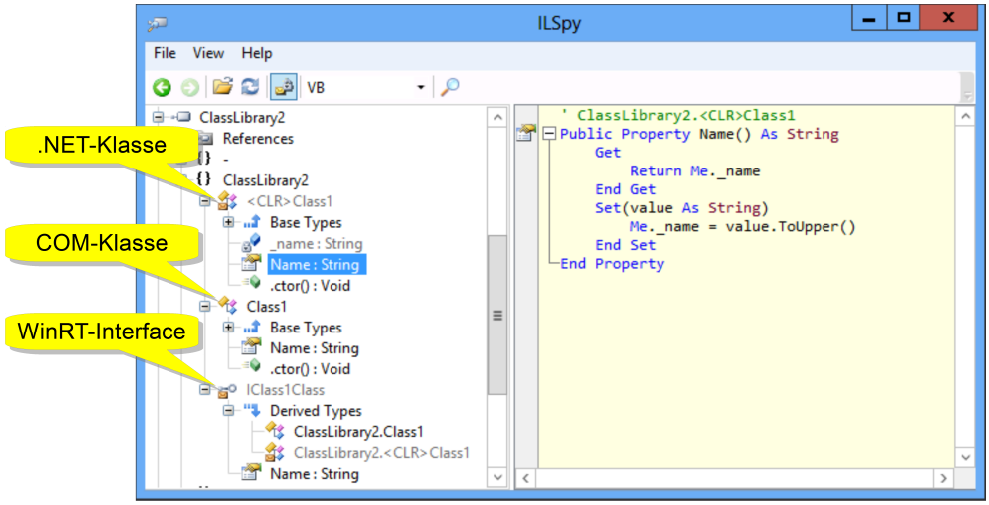
- Öffentliche Klassen müssen *NotInheritable* sein.
- Öffentliche *Structures* dürfen keine Methoden, Ereignisse, Konstruktoren definieren.

[1] Das Ende der Registry wurde zwar von diversen Gurus schon vor einiger Zeit verkündet, aber Totgesagte leben bekanntlich länger!

- 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 (Windows Store-Apps) und stellen als Ausgabetyp "Komponente für Windows Runtime" ein. Erstellen Sie jetzt das Projekt, wird statt einer 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.

18.5 Praxisbeispiel

18.5.1 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 8/10),
- 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 oder 10 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>
        <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 **.vbproj* 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:

```
...
Imports Windows.Devices.Enumeration

Class MainWindow
...
```

Mit dem Klick auf die Schaltfläche soll die *TreeView* gefüllt werden. Da der WinRT-Aufruf asynchron ist, müssen wir die Methode mit *async* kennzeichnen:

```
    Private Async Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
```

Hier folgt der eigentliche WinRT-Aufruf:

```
        Dim dicol As DeviceInformationCollection = 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 *For Each*-Anweisungen:

```
        For Each di As DeviceInformation In dicol.OrderBy(Function(d) d.Name)
```

Der jeweilige Haupteintrag:

```
            Dim tvi As New TreeViewItem() With {.Header = di.Name}
```

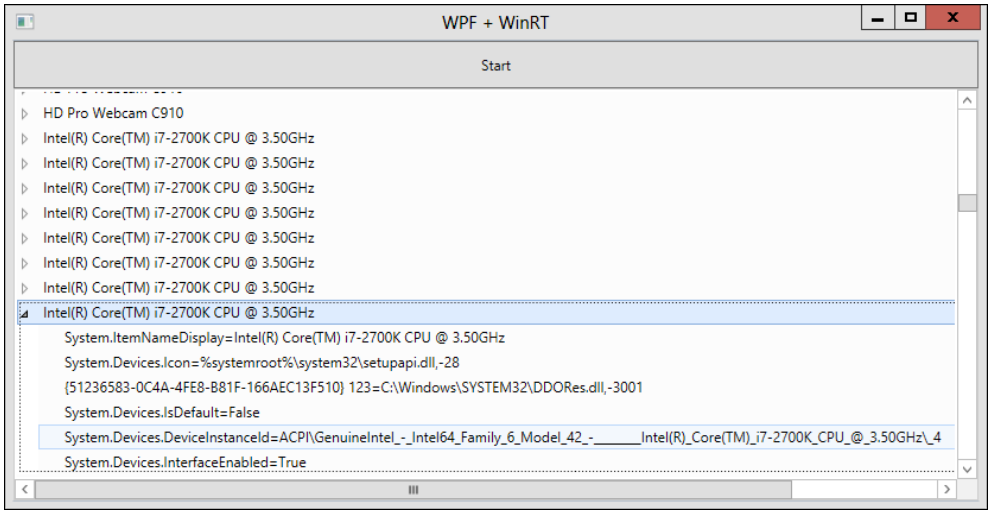
Hier lesen wir noch die Eigenschaften aus, diese befinden sich in einer extra Collection:

```
            For Each prop In di.Properties
                tvi.Items.Add(New TreeViewItem() With {.Header = _
                    Convert.ToString(prop.Key) & "=" & Convert.ToString(prop.Value)})
            Next
            TreeView1.Items.Add(tvi)
        Next
    End Sub
End Class
```

Das war es auch schon, 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:



Bemerkungen

- 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.
- Natürlich könnten Sie diese Liste mit LINQ noch filtern, aber das soll nicht Gegenstand dieses Beispiels sein.

Kapitel 19

App-Oberflächen entwerfen

Windows ist zunächst einmal Oberfläche pur. Aus diesem Grund wollen wir uns in einem ersten Schritt mit den Entwurfskonzepten und den wichtigsten Elementen für App-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 App-Programmierung übertragen. Um uns nicht endlos zu wiederholen, verweisen wir deshalb auf die entsprechenden WPF-Kapitel.

19.1 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 VB-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 932) bewundern durften und wenden uns ganz schlicht der Programmoberfläche und den Oberflächenkonzepten von Apps 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 Serpen-

[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.

tinen) 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.

19.1.1 XAML (oder HTML 5) für die Oberfläche

Sie haben es ja nun schon mehrfach gelesen: die Oberfläche Ihrer 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 VB-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.vb*-Dateien.

Sehen wir uns also eine dieser XAML-Dateien näher an.

BEISPIEL 19.1: 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 ist besonders wichtig, es handelt sich um einen Verweis auf den aktuelle Projekt:

```
    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 19.1: 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>
```

19.1.2 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.vb* schnell beantwortet, denn hier findet sich der schon vorbereitete Eintrittspunkt in unser Programm:

BEISPIEL 19.2: Eintrittspunkt in *App.xaml.vb*

VB

```
...
    Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
        Dim rootFrame As Frame = Window.Current.Content
        If rootFrame Is Nothing Then
            rootFrame = New Frame()
            If args.PreviousExecutionState = ApplicationExecutionState.Terminated Then
            End If
            Window.Current.Content = rootFrame
        End If
        If rootFrame.Content Is Nothing Then
            If Not rootFrame.Navigate(GetType(MainPage), args.Arguments) Then
                Throw New Exception("Failed to create initial page")
            End If
        End If
        Window.Current.Activate()
    End Sub
...
```

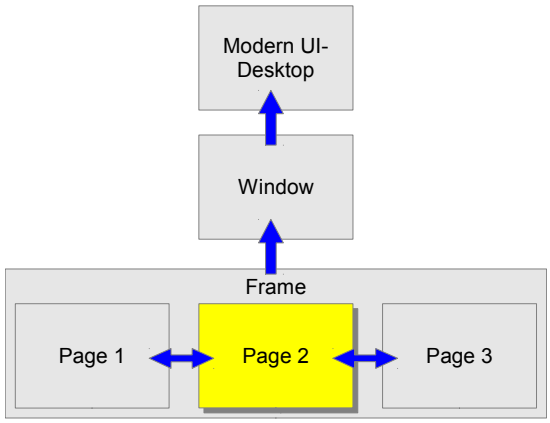
Folgendes passiert:

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).

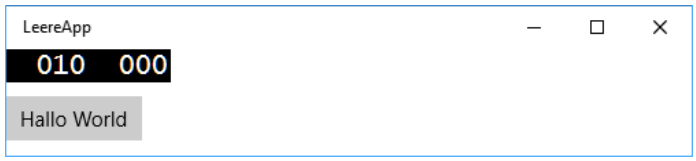
Die folgende Abbildung zeigt noch einmal den Sachverhalt:



Wenn Sie dieses Konzept verstanden haben, wird Ihnen auch klar, warum folgender Code problemlos funktioniert:

```
    Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
...
            Window.Current.Content = New Button() With {.Content = "Hallo World"}
...
            Window.Current.Activate()
    End Sub
```

In diesem Fall wird eine App 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.



19.1.3 Das Befehlsdesign

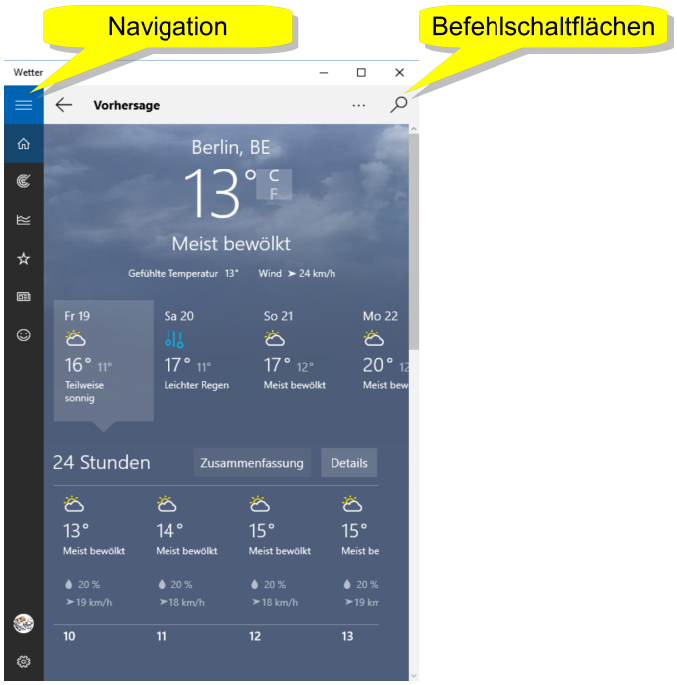
Bevor Sie jetzt anfangen, frei nach MUPS[1] Ihre Oberflächen zu entwerfen, sollten Sie sich zunächst mit einigen grundlegenden Design-Richtlinien für die Gestaltung der App-Oberflächen vertraut machen:

HINWEIS: Die App-Oberfläche sollte sich grundsätzlich auf die wesentlichsten Gestaltungselemente beschränken, der Content/die Information steht im Mittelpunkt.

Doch im Gegensatz zur vorhergehenden Version unter Windows 8, wo es das Konzept der App-Leisten am oberen und unteren Bildschirmrand gab, wird jetzt alles wieder anders und komplizierter. Der Grund ist die Unterstützung für Windows-Phone und damit einhergehend ein eher senkrecht orientierter Bildschirm mit Toucheingabe, der andere Erfordernisse an die Anwendungsergonomie stellt, als eine reine Desktop-App.

Befehlsleisten

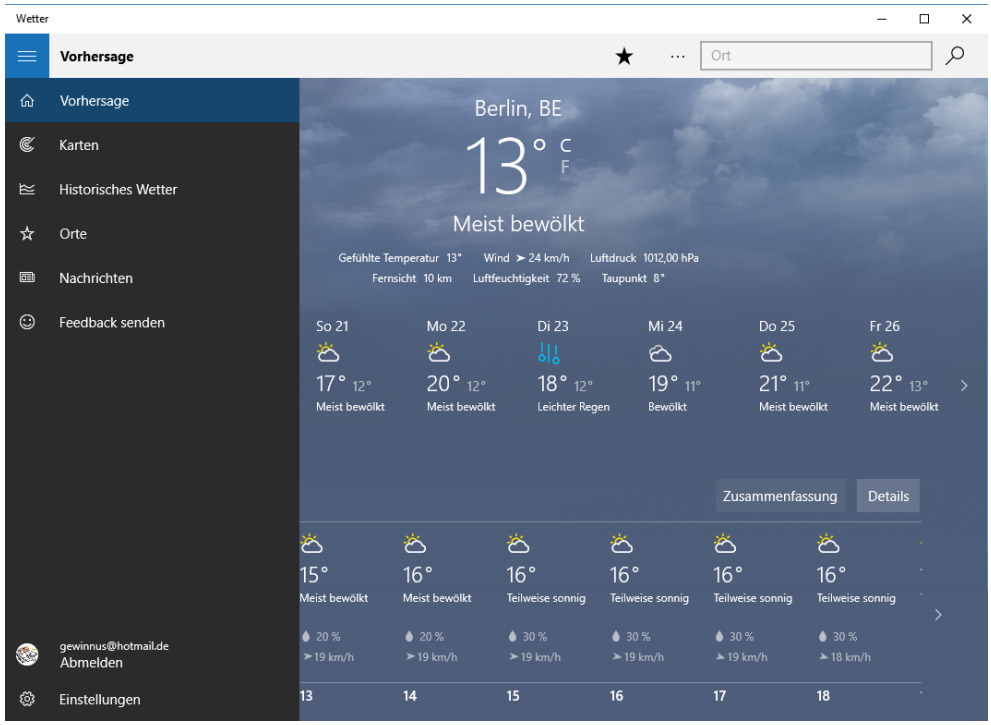
Für optionale Steuerelemente (Konfiguration, Navigation etc.) stehen Ihnen die Befehlsleisten am oberen und linken Bildschirmrand zur Verfügung. Diese werden optional eingeblendet und im Normalfall automatisch ausgeblendet, wenn der Nutzer eine Aktion ausgewählt hat.



[1] Methode des Unbekümmerten Probierens

Die App-Leisten werden entweder per Code, per rechter Maustaste oder mittels Wischbewegung der Finger angezeigt.

Die folgende Abbildung zeigt die Wetter-App mit der eingeblendeten Navigationsleiste am linken Rand:



Wie Sie sehen, sind die enthaltenen Schaltflächen nicht irgendwo angeordnet, sondern an der Außenseite. 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 40" 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.

HINWEIS: Da Windows 10 auf die Charmbar verzichtet, steht der linke/rechte Rand für Befehlsleisten wieder zur Verfügung.

Weitere Informationen finden Sie unter der folgenden Adresse, Microsoft hat sich da schon reichlich Gedanken zum Thema gemacht:

LINK: `https://dev.windows.com/en-us/design`

HINWEIS: Wie Sie die Leisten (*ApplicationBar*) in Ihren Apps selbst erzeugen und nutzen, zeigt Abschnitt 22.3.6 ab Seite 1133.

Settingsbereich

Auch hier dürfen bisherige Windows 8 App-Entwickler umdenken, durch den Entfall der Charmbar werden die Programmeinstellungsoptionen ab sofort wieder in der App selbst anzeigt. Dies ist auch sinnvoll, da gerade bei großen Desktops der Abstand zwischen frei beweglicher App und rechtem Bildschirmrand beträchtlich sein kein.

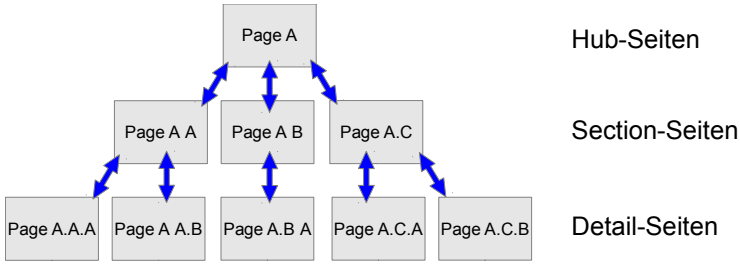
Auf eine finale Lösung hat sich Microsoft bei den eigenen Apps noch nicht geeinigt, die Mehrzahl blendet die Schaltfläche für die Einstellungen am linken unteren Rand der App ein, die einzelnen Gruppen innerhalb der Einstellungen werden per Pivot-Steuerlement dargestellt.

19.1.4 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.

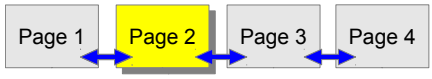
Da wir nur eine Seite innerhalb unserer App darstellen können, müssen wir uns klar darüber werden, in welcher Reihenfolge zwischen den Seiten gewechselt werden kann. Dabei bieten sich zwei grundsätzliche Varianten an:

- Hierarchische Organisation



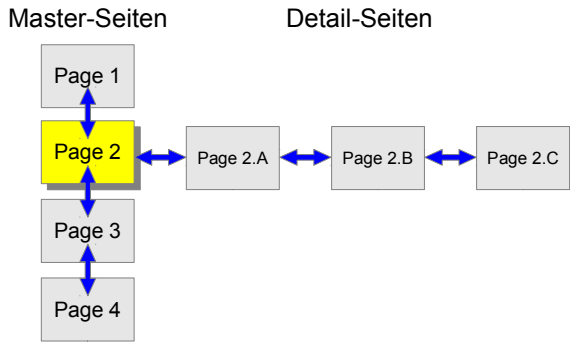
  Diese entspricht dem Aufbau der meisten Nachrichten-Webseiten, von einer Startseite werden Sie meist zu den Rubriken und von dort zu den einzelnen Artikeln weitergeleitet.

- Flache Organisation



  Diese Variante ist mit der Navigation innerhalb von Blogs oder News-Listen vergleichbar. Optional ist der Sprung auf die erste Seite denkbar.

- Master-Detail-Organisation



  Diese Variante finden Sie beispielsweise in der aktuellen Finanz-App. Links werden die Hauptgruppen angezeigt, rechts kann über ein Pivot-Control zwischen den Detailseiten geblättert werden.

Je nach Organisationstyp wählen Sie später den Projekttyp und die weiteren Detailseiten in Visual Studio aus.

Mehr dazu unter

LINK: `https://msdn.microsoft.com/library/windows/apps/dn958438.aspx`

19.1.5 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 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 Windows 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.

19.1.6 Verwendung von Schriftarten

Verwenden Sie grundsätzlich so wenig Schriftstile wie möglich, und halten Sie sich an die Vorgaben von Microsoft

Siehe dazu auch:

LINK: `https://msdn.microsoft.com/de-de/library/windows/apps/hh700394.aspx`

19.2 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 19.1.2 beschrieben.

19.2.1 Die Startseite festlegen

In den bisherigen Beispielen hatte wir es schon mehrfach erwähnt, der Eintrittspunkt für Ihre App liegt in der Datei *App.xaml.vb*, genauer in der Methode *OnLaunched*.

```
Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
    Dim rootFrame As Frame = Window.Current.Content
    If rootFrame Is Nothing Then
        rootFrame = New Frame()
        If args.PreviousExecutionState = ApplicationExecutionState.Terminated Then
        End If
        Window.Current.Content = rootFrame
    End If
```

An dieser Stelle legen Sie die Startseite fest:

```
    If rootFrame.Content Is Nothing Then
        If Not rootFrame.Navigate(GetType(MainPage), args.Arguments) Then
            Throw New Exception("Failed to create initial page")
        End If
    End If
    Window.Current.Activate()
End Sub
```

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.

19.2.2 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 *Navigat*e-Methode aufrufen.

BEISPIEL 19.3: Seitenwechsel als Reaktion auf das *Click*-Ereignis

VB

```
Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
    Me.Frame.Navigate(GetType(BasicPage2))
End Sub
```

Eine Überladung der *Navigate*-Methode ermöglicht zusätzlich die Übergabe von Parametern, dies können einfache Eigenschaften oder auch gleich die aktuelle Page-Instanz sein.

BEISPIEL 19.4: Parameterübergabe

VB

```
Me.Frame.Navigate(GetType(BasicPage2), TextBox1.Text)
```

oder auch

```
Me.Frame.Navigate(GetType(BasicPage2), Me)
```

Die Auswertung erfolgt in der *OnNavigatedTo*-Methode der aufgerufenen Page:

```
Protected Overrides Sub OnNavigatedTo(e As Navigation.NavigationEventArgs)
    Dim inhaltTextbox As String = TryCast(e.Parameter, String)
```

oder auch

```
    Dim page1 As BasicPage1 = TryCast(e.Parameter, BasicPage1)
```

HINWEIS: Vergessen Sie nicht, dass Sie für den externen Zugriff auf die Seitenelemente zunächst auch den *FieldModifier* anpassen müssen.

BEISPIEL 19.5: Änderung *FieldModifier* für eine *TextBox*

VB

```
<TextBox Text="Hallo" Name="TextBox1" x:FieldModifier="public"></TextBox>
```

Ein komplettes Beispiel finden Sie im Praxisbeispiel

► 19.5.1 Seitennavigation und Parameterübergabe

19.2.3 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*:

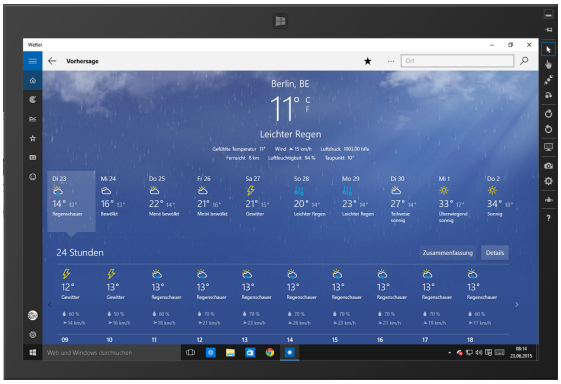
```
<common:LayoutAwarePage
    x:Name="pageRoot"
    x:Class="NavigationParameter.BasicPage1"
...
    mc:Ignorable="d" NavigationCacheMode="Enabled" >
...
```

19.3 App-Darstellung

Im Gegensatz zu reinen Desktop-Anwendungen müssen Sie sich bei einer App auf einige Besonderheiten im Zusammenhang mit der Darstellung vorbereiten und entsprechend darauf reagieren.

19.3.1 Vollbild quer und hochkant

Die beiden folgenden Abbildungen zeigen zwei Zustände einer App, die im Vollbildmodus dargestellt wird[1] (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.

Im Gegensatz zu Windows 8 hat Microsoft mit 8.1 die Anforderungen an die Bildschirmauflösung weiter gesenkt, um sich auch den Markt der Billiggeräte (Tablets) zu erschließen. Derzeit ist von einer minimalen Auflösung von 1024x600 Pixeln die Rede. Hier stellt sich allerdings die Frage, ob Ihre App in dieser Mini-Auflösung noch alle relevanten Informationen darstellen kann. Gerade der "Spagat" zwischen Riesen-Auflösung (4K) und o.g. Werten dürfte das Problem verdeutlichen.

Wie Sie auf den Formatwechsel reagieren können, zeigen wir Ihnen im Abschnitt 19.3.3, zunächst sehen wir uns noch die beiden anderen Zustände der App an.

[1] Wir verwenden dafür den Simulator.

19.3.2 Was ist mit Andocken und Füllmodus?

Was unter Windows 8 noch das Non plus ultra war, gemeint ist die zwangsweise Vollbilddarstellung und das damit verbundene Andocken, ist zwei Versionen weiter schon wieder vergessen. Windows 8.1 hatte es ja schon angekündigt, hier waren die Apps im Docking-Modus frei skalierbar geworden.

Gehen Sie also jetzt davon aus, dass sich eine Windows 10 App wie eine ein normale Desktop-Anwendung verhält, die optional angedockt werden kann. Damit stoßen Sie jetzt allerdings auf ein Problem, dass es auch früher schon gab: die minimale Größe der App. Nichts leichter als dies, Sie können diese über das *SizeChanged*-Ereignis auswerten.

19.3.3 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.

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.

Im Gegensatz zur vorherigen Version bzw. zu den Windows 8-Apps sollten Sie auf die Verwendung von *Windows.UI.ViewManagement.ApplicationView.Value* verzichten, da ja die App mittlerweile frei skaliert werden kann.

BEISPIEL 19.6: Reaktion auf Vergrößerung der App

VB
```
...
    Private Async Sub Page_SizeChanged(sender As Object, e As SizeChangedEventArgs)
        If e.NewSize.Width > e.PreviousSize.Width Then
            Dim messageDialog = New MessageDialog("Größenänderung")
            Await messageDialog.ShowAsync()
        End If
    End Sub
...
```

VisualStateManager

Nicht immer wollen oder können Sie gleich den kompletten Inhalt Ihrer Seite ändern, nur weil die App skaliert 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 Erreichen einer bestimmten Formularbreite auf die Änderung reagiert werden. Der Vorteil dieser Vorgehensweise: Sie können im XAML-Code die "Ursache" (StateTrigger) und die "Wirkung" (Setter) definieren. Dies ist auch im Hinblick auf viele XAML-Eigenschaften sinnvoll, da sich diese per VB-Code meist recht umständlich setzen lassen

BEISPIEL 19.7: Der *VisualStateManager* in Aktion

XAML

```
    <Grid Name="Grid1" Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup>
```

Der *State* "winBreit":

```
                <VisualState x:Name="winBreit">
                    <VisualState.StateTriggers>
```

Auslöser ist eine Fensterbreite > 720:

```
                        <AdaptiveTrigger MinWindowWidth="720" />
                    </VisualState.StateTriggers>
```

Die Reaktion:

```
                    <VisualState.Setters>
```

Wir ändern die Beschriftung eines TextBlocks:

```
                        <Setter Target="textBlock1.Text" Value="Fenster breit"/>
                    </VisualState.Setters>
                </VisualState>
```

Der Normalzustand:

```
                <VisualState x:Name="winSchmal">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="0" />
                    </VisualState.StateTriggers>
                    <VisualState.Setters>
                        <Setter Target="textBlock1.Text" Value="Fenster schmal"/>
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <TextBlock x:Name="textBlock1" HorizontalAlignment="Left" Margin="48,34,0,0"
                   TextWrapping="Wrap" Text="TextBlock" VerticalAlignment="Top"/>
```

BEISPIEL 19.7: Der *VisualStateManager* in Aktion

```
    </Grid>
</Page>
```

Wenn Sie den kompletten *State* "winSchmal" auskommentieren, kehrt die Oberfläche bei einer Breite kleiner als 720 zur ursprünglichen Definition zurück, die Sie im XAML-Editor definiert haben.

19.4 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 10 eine Skalierung in folgenden Prozentstufen:

- 100%
- 120%
- 140% auf HD-Tablets
- 150% und 160% (Windows Phone)
- 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 *DisplayInformation* (*DisplayProperties* ist veraltet).

BEISPIEL 19.8: Auswerten von Skalierungsänderungen und Anzeige der aktuellen Skalierung in einem *TextBlock*.

VB

```
...
Public NotInheritable Class MainPage
    Inherits Page

    Private Sub Page_Loaded(sender As Object, e As RoutedEventArgs)
        AddHandler DisplayInformation.GetForCurrentView().DpiChanged, AddressOf DpiAenderung
    End Sub

    Private Sub DpiAenderung(sender As DisplayInformation, args As Object)
        tb1.Text = "ResolutionScale: " &
                   DisplayInformation.GetForCurrentView().ResolutionScale.ToString()
        tb2.Text = "LogicalDpi: " &
                   DisplayInformation.GetForCurrentView().LogicalDpi.ToString()
    End Sub
End Class
```

Während *ResolutionScale* die Werte 100, 120, 140 oder 180 zurückgibt, liefert *LogicalDpi* die Auflösung des Displays (bei einem Desktopmonitor 96).

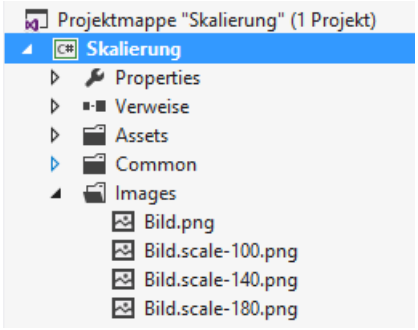
HINWEIS: *ResolutionScale* entspricht **nicht** den Einstellungen in der Systemsteuerung, diese wirken sich nur auf den Desktop und nicht auf Apps aus.

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 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:

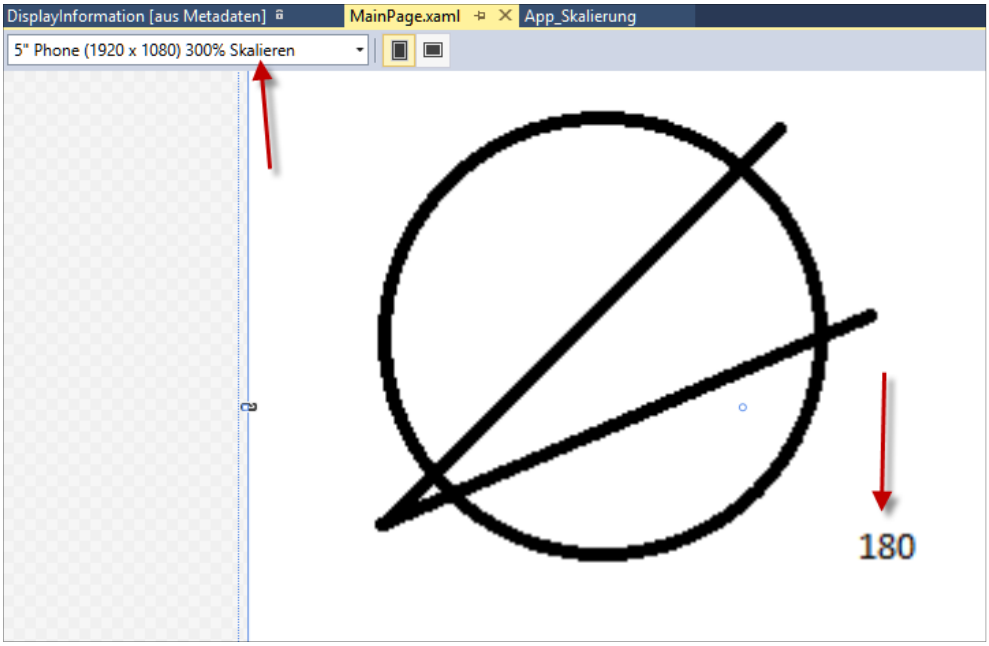
```
<Bildname>.scale-100.<Extension>
<Bildname>.scale-140.<Extension>
<Bildname>.scale-180.<Extension>
```

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 im Designer in der linken oberen Ecke die verschiedenen Auflösungen mit den jeweiligen Skalierungswerten durch testen (zur Entwurfszeit)



Im obigen Beispiel wurde für 300% Skalierung die bestmöglich vorliegende Grafik *Bild.scale-180.png* ausgewählt.

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.

19.5 Praxisbeispiele

19.5.1 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

```
...
        <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 Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Me.Frame.Navigate(GetType(BasicPage2), Me)
    End Sub
```

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

```
...
    <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}"/>
        </StackPanel>
    </ScrollViewer>
...
```

[1] Ohne diese Erweiterung kann von anderen Seiten nicht auf die *TextBox* zugegriffen werden.

Quellcode BasicPage2

Auswertung der Parameter in der *OnNavigatedTo*-Methode:

```
    Protected Overrides Sub OnNavigatedTo(e As Navigation.NavigationEventArgs)
        Dim page1 As BasicPage1 = TryCast(e.Parameter, BasicPage1)
        textBlock1.Text = page1.TextBox1.Text
        MyBase.OnNavigatedTo(e)
    End Sub
```

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.

19.5.2 Die Fensterkopfzeile anpassen

Mit Windows 10 handelt es sich auch bei Apps wieder um echte "Fenster", die wiederum über eine Kopfzeile verfügen. Zwangsläufig kommt nun auch der Wunsch auf, diese Kopfzeile anzupassen.

Zwei wesentliche Varianten bieten sich an:

- Anpassen der Vorder- und Hintergrundfarben sowie der Beschriftung
- Teilweises Ausblenden der Kopfzeile (die Schaltflächen bleiben erhalten)

Unser kleines Beispielprogramm zeigt Ihnen die Umsetzung.

Oberfläche

Erstellen Sie eine neue Windows App und fügen Sie lediglich zwei Schaltflächen in die MainPage ein:

```
...
    <Grid Background="Yellow">
        <StackPanel>
            <Button Click="Button_Click">Text im Titel und Farbe</Button>
```

```
            <Button Click="Button_Click_1">Ohne Titel</Button>
        </StackPanel>
    </Grid>
...
```

Quellcode

```
...
Imports Windows.ApplicationModel.Core
Imports Windows.UI
...
Public NotInheritable Class MainPage
    Inherits Page
...
```

Kopfzeile Text und Farben verändern:

```
    Private Sub Button_Click(sender As Object, e As RoutedEventArgs)
        ApplicationView.GetForCurrentView().Title = "Meiner neuer Titel"
        ApplicationView.GetForCurrentView().TitleBar.BackgroundColor = Colors.Red
        ApplicationView.GetForCurrentView().TitleBar.ButtonBackgroundColor = Colors.Green
    End Sub
```

Kopfzeile entfernen (Schaltflächen bleiben erhalten):

```
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        CoreApplication.GetCurrentView().TitleBar.ExtendViewIntoTitleBar = True
    End Sub
End Class
```

HINWEIS: Die Eigenschaft *ExtendViewIntoTitleBar* steht nur über das *CoreApplication*-Objekt zur Verfügung (siehe oben).

Test

Die Standardansicht:

Farbliche Änderung des Kopfes:

Ausblenden des Kopfes:

Kapitel 20

Die wichtigsten Controls

Nach einem ersten Rundflug über das Erstellen von 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 bzw. 10 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.

20.1 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 (*Page*) finden Sie bereits ein zentrales *Grid* vor, das diese Aufgabe übernimmt. Weitere Informationen zu den Layout-Controls finden ab Seite 987.

20.1.1 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 20.1: Einfache Textausgabe mit vordefiniertem Style

XAML

```
...
   <TextBlock Text="TextBlock einfach" Style="{StaticResource BasicTextStyle}" />
...
```

BEISPIEL 20.2: *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!" />
```

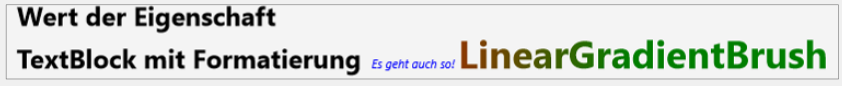
Auch komplexe Füllpinsel lassen sich definieren:

```
<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 20.3: 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:

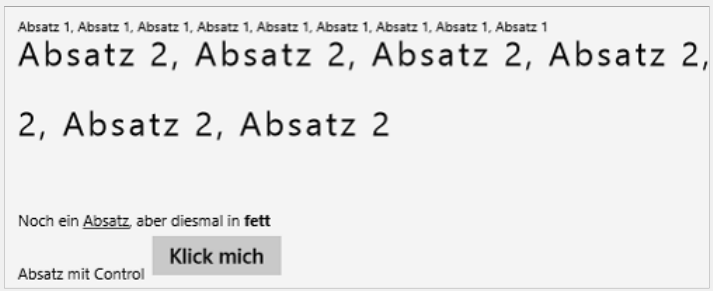
```
             <Paragraph>Noch ein <Underline>Absatz</Underline>, aber diesmal in
                   <Bold>fett</Bold>
             </Paragraph>
             <Paragraph>Absatz mit Control
```

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 20.4: Zeilen zur Laufzeit einfügen

VB
```
Imports Windows.UI.Xaml.Documents
...
        Dim run As New Run() With {.Text = "Text zur Laufzeit! Text zur Laufzeit!"}
        Dim para As New Paragraph()
        para.Inlines.Add(run)
        para.Inlines.Add(New LineBreak())
        para.Inlines.Add(New Run() With {.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.

20.1.2 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 20.5: 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 20.5: 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>
...
```

VB

Die Funktionalität des *HyperlinkButton* muss erst programmiert werden:

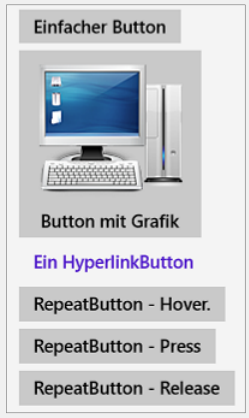
```
...
   Private Async Sub HyperlinkButton_Click_1(sender As Object, e As RoutedEventArgs)
        Dim uri As New Uri("http://www.doko-buch.de")
        Dim success As Boolean = Await Windows.System.Launcher.LaunchUriAsync(uri)
    End Sub
```

Die Reaktion der *RepeatButton*-Controls:

```
    Private Sub RepeatButton_Click_1(sender As Object, e As RoutedEventArgs)
        TryCast(sender, RepeatButton).Content += "."
    End Sub
...
```

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 immer an "Fast and Fluid" (siehe Seite 921).

20.1.3 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 20.6: 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 20.6: Verwendung von *CheckBox, RadioButton, ToggleButton, ToggleSwitch*

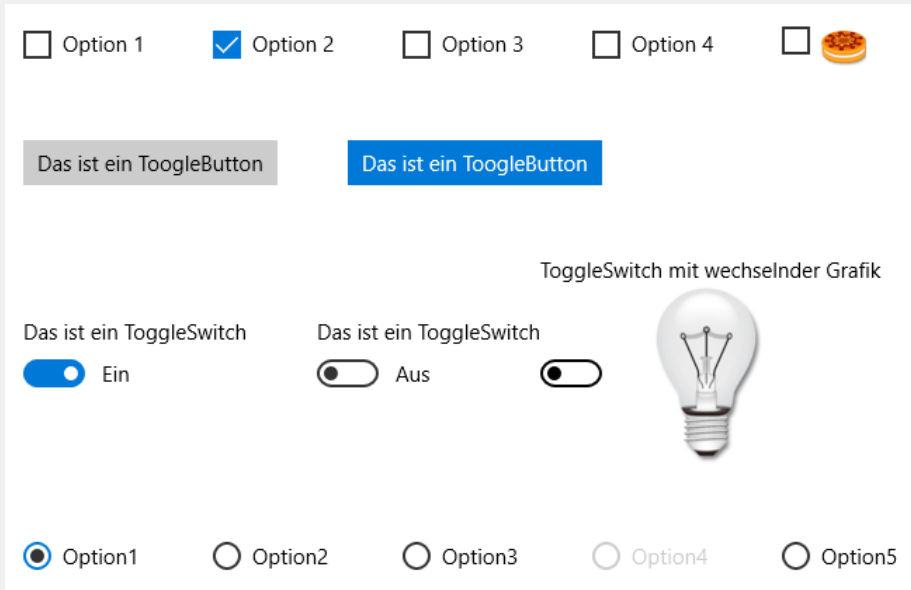
```
                    <ToggleSwitch.OffContent>
                        <Image Source="Images/bulboff.png"/>
                    </ToggleSwitch.OffContent>
                </ToggleSwitch>
            </StackPanel>
```

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:



20.1.4 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 20.7: *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>
...
```

VB

```
...
```

Wir laden eine Text- oder Richtext-Datei in die *RichEditBox*:

```
    Private Async Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
```

Das Öffnen der "Dateidialoge" – jetzt *FileOpenPicker* – erfordert die Vollbildansicht:

```
        If ApplicationView.Value <> ApplicationViewState.Snapped Then
            Dim open As New Windows.Storage.Pickers.FileOpenPicker()
            open.SuggestedStartLocation = _
                          Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary
            open.FileTypeFilter.Add(".rtf")
            open.FileTypeFilter.Add(".txt")
            Dim file As Windows.Storage.StorageFile = Await open.PickSingleFileAsync()
```

BEISPIEL 20.7: *TextBox, PasswordBox, RichEditBox*

Wurde eine Datei ausgewählt ...

```
            If file IsNot Nothing Then
```

... können wir diese in die *RichEditBox* laden:

```
                Dim strm As Windows.Storage.Streams.IRandomAccessStream = _
                     Await file.OpenAsync(Windows.Storage.FileAccessMode.Read)
                reb1.Document.LoadFromStream(Windows.UI.Text.TextSetOptions.FormatRtf, strm)
            End If
        End If
    End Sub
```

Alternativ können Sie die Inhalte der *RichEditBox* auch per Code erzeugen:

```
    Private Async Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
```

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, vbLf & "Ein 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"""
```

Hier fügen wir eine Grafik ein, die der Nutzer per *FileOpenPicker* auf dem System auswählt:

```
        Dim open As New Windows.Storage.Pickers.FileOpenPicker()
        open.SuggestedStartLocation = _
```

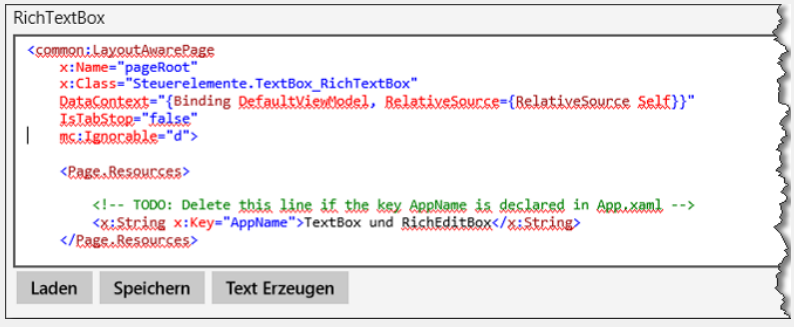
BEISPIEL 20.7: ***TextBox, PasswordBox, RichEditBox***

```
                    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")

        Dim file As Windows.Storage.StorageFile = Await open.PickSingleFileAsync()
        Dim fileStream As Windows.Storage.Streams.IRandomAccessStream = _
                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)
    End Sub
```

Die Inhalte der *RichEditBox* speichern:

```
    Private Async Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        If ApplicationView.Value <> ApplicationViewState.Snapped Then
            Dim open As New Windows.Storage.Pickers.FileSavePicker()
            open.SuggestedStartLocation = _
                    Windows.Storage.Pickers.PickerLocationId.DocumentsLibrary
            open.FileTypeChoices.Add("Rich Text Format", New String() {".rtf"})
            Dim file As Windows.Storage.StorageFile = Await open.PickSaveFileAsync()
             If file IsNot Nothing Then
                Dim strm As Windows.Storage.Streams.IRandomAccessStream = _
                       Await file.OpenAsync(Windows.Storage.FileAccessMode.ReadWrite)
                reb1.Document.SaveToStream(TextGetOptions.FormatRtf, strm)
                strm.Dispose()
             End If
        End If
    End Sub
...
```

Ergebnis



20.1.5 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 20.8: *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>
    </Image>
</Canvas>
...
```

VB

Wie versprochen, müssen wir uns noch um die eigentliche Manipulation kümmern, wir hatten lediglich die Reaktion darauf "freigeschaltet":

```
Private Sub Image_ManipulationDelta(sender As Object,
                                    e As ManipulationDeltaRoutedEventArgs)
```

Aktuelle Einstellungen abrufen:

```
Dim comptrans As CompositeTransform = DirectCast(TryCast(sender,
                        Image).RenderTransform, CompositeTransform)
```

BEISPIEL 20.8: *Image*-Control und Gestensteuerung

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
    End Sub
...
```

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.

20.1.6 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 20.9: ***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,
                           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):

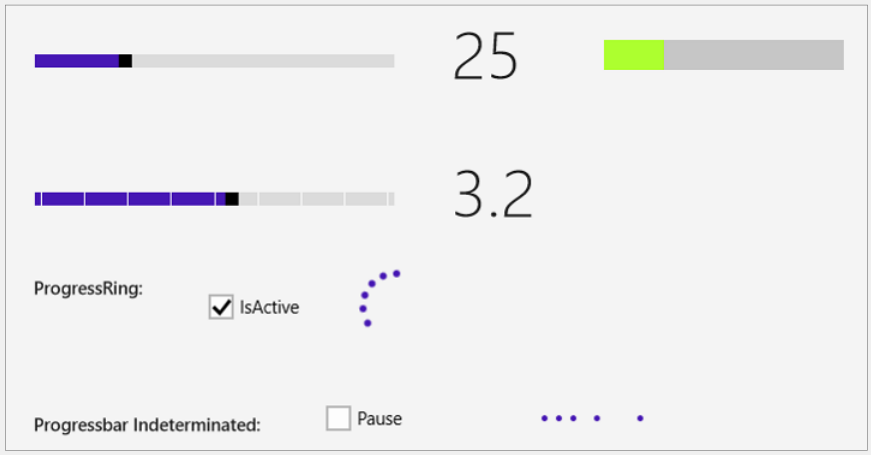
```
             <ProgressBar Maximum="100" Value="{Binding ElementName=Slider1, Path=Value}"
                         IsIndeterminate="True" Height="25" Width="200"
```

BEISPIEL 20.9: *Slider, ProgressBar, ProgressRing*

```
                        Foreground="GreenYellow"
                        ShowPaused="{Binding IsChecked, ElementName=Check1}"/>
        </StackPanel>
...
```

Ergebnis

Verschieben Sie die beiden *Slider*:



20.1.7 Border, Ellipse, Rectangle

An dieser Stelle erwartet den WPF-Programmierer nichts Neues, die Controls verfügen lediglich über die zusätzlichen Events für die Toucheingabe (*Tapped* ...)

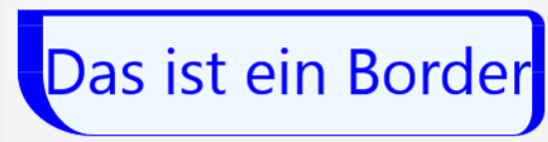
BEISPIEL 20.10: Verwendung *Border*

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>
...
```

Ergebnis

Die Laufzeitansicht:



20.2 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.

20.2.1 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 20.11: Verwendung *Canvas*

VB

```
...
    <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 20.12: Frei positionierter *TextBlock*

XAML

```
...
        <!-- Back Button and page title -->
        <Grid>
            <Grid.ColumnDefinitions>
```

BEISPIEL 20.12: Frei positionierter *TextBlock*

```
            <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

20.2.2 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.

20.2.3 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 20.13: Verwendung *ScrollViewer*

XAML

```
...
    <ScrollViewer Height="200" Width="200" HorizontalScrollBarVisibility="Auto" >
      <Image Source="Images/bulbon.png" Width="400" Height="400" />
    </ScrollViewer>
...
```

Ergebnis

20.2.4 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 20.14: 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.

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>
```

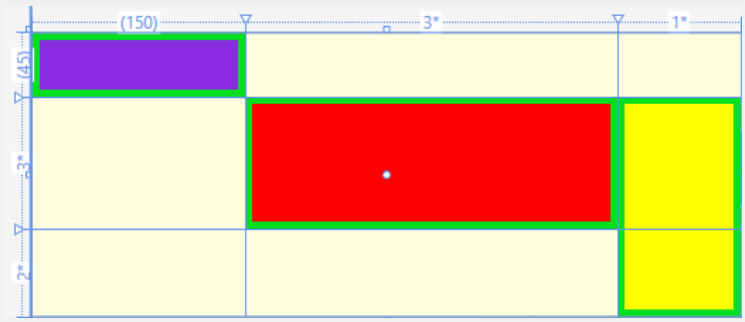
Beachten Sie das dritte Rechteck: Dieses erstreckt sich wegen *Grid.RowSpan="2"* über zwei Zeilen.

```
</Grid>
```

...

BEISPIEL 20.14: Verwendung *Grid*

Ergebnis



20.2.5 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 20.15: 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>
    <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>
```

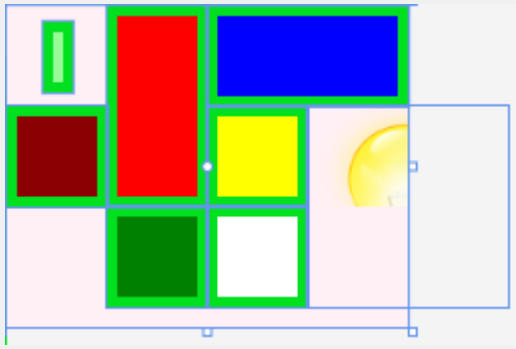
Das Image wird beschnitten, da zu groß:

```
    <Image Source="Images/bulbon.png" Height="100" Width="100"/>
</VariableSizedWrapGrid>
...
```

BEISPIEL 20.15: Verwendung *VariableSizedWrapGrid*

Ergebnis

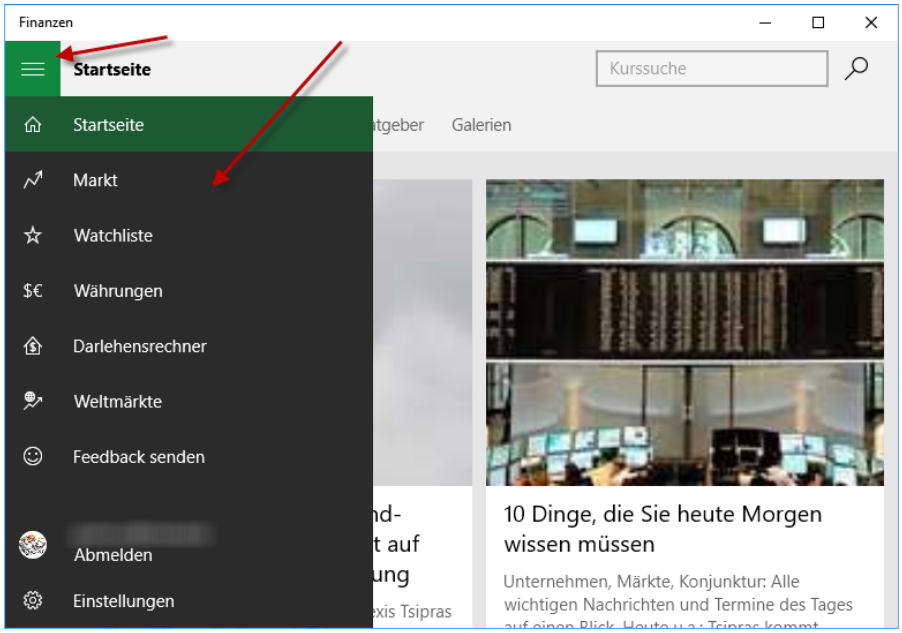
Die Entwurfsansicht:



20.2.6 SplitView

Sicher wird Ihnen eine der neuen Gestaltungsrichtlinien von Windows 10, die bei fast allen mitgelieferten Apps zum Einsatz kommt, nicht verborgen geblieben. Die Rede ist vom aufklappbaren Menü am linken App-Rand, das entweder über dem Content oder neben dem Content der Page eingeblendet wird, bis eine Option gewählt ist. Hierbei handelt es sich um die für Windows-Programmierer neue *Splitview*.

Ein gutes Beispiel ist die Finanz-App von Windows 10, die über eine *SplitView* die einzelnen Aufgaben (*Startseite, Markt, Watchliste ...*) bereitstellt:



Wir könnten jetzt auch ein komplexes Beispiel entsprechend den Microsoft-Beispielen darstellen, denken aber, dass eine auf das wesentliche reduzierte Lösung besser verständlich ist.

BEISPIEL 20.16: Verwendung *SplitView*

XAML

...

Das umgebende *Grid* aus der *Page*:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
```

Wir definieren zwei Spalten (einmal die Spalte mit der Ein-/Ausschaltfläche) und dann der eigentlich sichtbare Bereich:

```
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
```

Über den *VisualStateManager* wechseln wir zwischen zwei Modi der *Splitview*: Ist der Bildschirm schmaler als 641 Pixel, wird die Menüauswahl **über** dem Content angezeigt, anderenfalls **links daneben**:

```
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="wideState">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="641" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="Splitter.DisplayMode" Value="Inline"/>
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="narrowState">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="Splitter.DisplayMode" Value="Overlay"/>
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
```

Hier die eigentliche *SplitView*:

```
    <SplitView x:Name="Splitter" IsPaneOpen="True" Grid.Column="1">
```

Zunächst der optional sichtbare Navigationsbereich:

```
        <SplitView.Pane>
            <RelativePanel Background="{ThemeResource
                                        SystemControlBackgroundChromeMediumBrush}">
```

BEISPIEL 20.16: Verwendung *SplitView*

Wir zeigen eine Überschrift und eine *ListBox* an, in der wiederum die einzelnen Menüpunkte dargestellt werden. Der Vorteil dieser Variante: ist der Bildschirm nicht hoch genug, erscheint für die *ListBox* ein *Scrollbar*, die Menüpunkte sind also alle erreichbar.

```
                <TextBlock x:Name="SampleTitle" Text="DOKO-Beispiel" TextWrapping="Wrap"
                           Margin="0,10,0,0"/>
                <ListBox x:Name="ListBox1"
                         SelectionChanged="ScenarioControl_SelectionChanged"
                         SelectionMode="Single" HorizontalAlignment="Left"
                         VerticalAlignment="Top"
                         RelativePanel.Below="SampleTitle" Margin="0,10,0,0" >
                </ListBox>
            </RelativePanel>
        </SplitView.Pane>
```

Kommen wir zum eigentlichen Content-Bereich, in dem später die jeweiligen Ansichten eingeblendet werden sollen:

```
        <RelativePanel>
```

Wir nutzen einen Frame, so könne wir beliebige Inhalte darstellen (Einzelzeiten):

```
            <Frame x:Name="ScenarioFrame" Margin="0,5,0,0" />
        </RelativePanel>
    </SplitView>
```

Was bleibt, ist die Darstellung der Spalte für die Ein-Aus-Schaltfläche:

```
    <Border Background="{ThemeResource SystemControlBackgroundChromeMediumBrush}"
            Padding="0,0,5,0">
```

Ein *ToggleButton* genügt uns, Standard ist (derzeit) ein "Hamburger"-Symbol, d h. drei Linien übereinander:

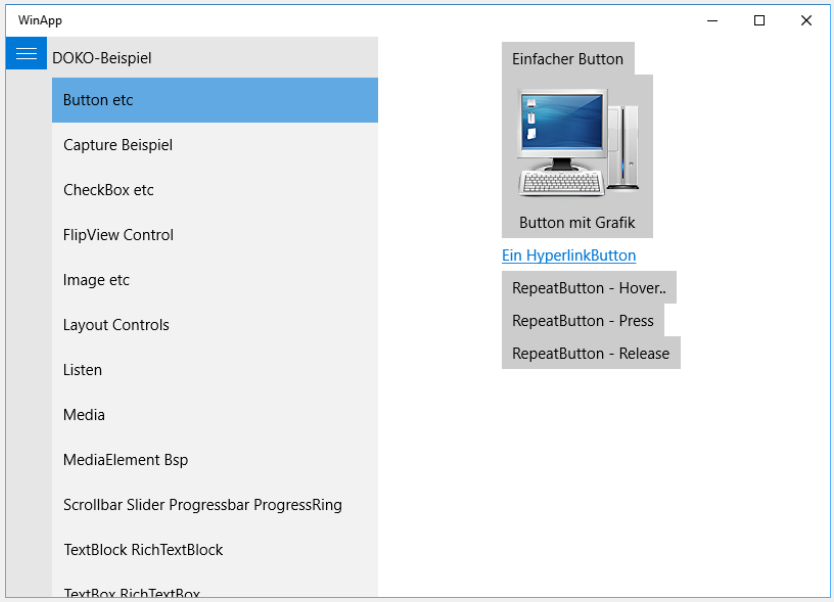
```
        <ToggleButton  Click="Button_Click" VerticalAlignment="Top"
             Foreground="{ThemeResource ApplicationForegroundThemeBrush}">
            <ToggleButton.Content>
                <FontIcon x:Name="Hamburger" FontFamily="Segoe MDL2 Assets"
                Glyph="&#xE700;" Margin="0,0,0,0"/>
            </ToggleButton.Content>
        </ToggleButton>
    </Border>
</Grid>
...
```

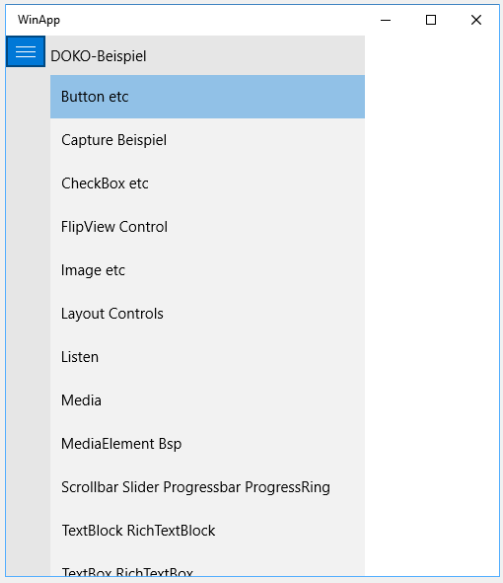
Alternativ können Sie die Schaltfläche natürlich auch in der Kopfzeile des Formulars unterbringen, so geht am linken Rand kein Platz verloren, was bei Smartphones etc. sicher von Interesse ist, fehlt es doch immer an Platz.

Ergebnis

Zunächst sehen Sie die Darstellung bei ausreichender Bildschirmbreite, nachfolgend die Overlay-Variante:

BEISPIEL 20.16: Verwendung *SplitView*





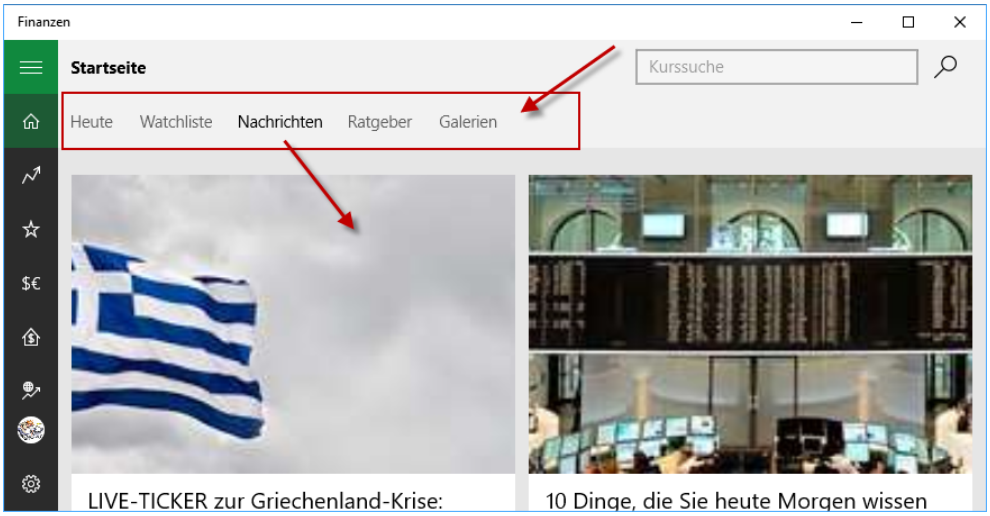
Bei der Overlay-Variante klappt das Menü nach der Auswahl wieder ein, so wird der darunterliegende Content sichtbar.

HINWEIS: Für die Navigation im Content-Bereich nutzen Sie am besten ein *Pivot*-Element.

Wer etwas mehr Aufwand investiert, kann natürlich auch in der Spalte für die Ein-/Aus-Schaltfläche Kurztasten bzw. Symbole einblenden, sodass die App auch dann bedienbar bleibt, wenn der Navigationsbereich eingeklappt ist. Hier nutzen Sie dann am besten ein *StackPanel*, ein Scrollbar würde wohl nur stören.

20.2.7 Pivot

Unter den Layout-Controls wollen wir Ihnen auch das Pivot-Control vorstellen. Dieses dient dazu, innerhalb einer Seite zwischen verschiedenen Optionen/Ansichten zu wechseln. Zum Beispiel werden bei der Finanz-App die einzelnen Rubriken *Heute, Watchliste, Nachrichten …* über ein Pivot-Control dargestellt:



Kommen wir zu einem kleinen Beispiel:

BEISPIEL 20.17: Verwendung *Pivot*

XAML

```
...
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

Das übergeordnete Pivot-Control:

```
        <Pivot>
```

Die einzelne Seite mit einem eigenen Header:

```
            <PivotItem Header="Pivot-Rubrik 1">
                <TextBlock>
                    Inhalte der ersten Seite
                </TextBlock>
            </PivotItem>
```

BEISPIEL 20.17: Verwendung *Pivot*

Die zweite Seite:

```
            <PivotItem Header="Pivot-Rubrik 2">
                <TextBlock>
                    Inhalte der zweiten Seite
                </TextBlock>
            </PivotItem>
```

Weitere Seiten ...

```
        </Pivot>
...
```

HINWEIS: Beachten Sie, dass innerhalb eines *PivotItems* ein Layout-Control zur Anordnung weiterer Steuerelemente benötigt wird.

20.2.8 RelativPanel

Ebenfalls neu für den Windows App-Entwickler ist das *RelativPanel*, das Sie beim Anordnen von Controls unterstützen soll. Wie der Name es schon andeutet, können Sie mit dessen Hilfe entweder ein Control bezüglich des *RelativPanels* ausrichten oder Sie nehmen Bezug auf ein anderes Control innerhalb des *RelativPanels*.

Für das Ausrichten bezüglich des *RelativPanels* stehen Ihnen folgende Eigenschaften zur Verfügung:

Eigenschaft	Standard
RelativePanel.AlignBottomWithPanel	*False*
RelativePanel.AlignHorizontalCenterWithPanel	*False*
RelativePanel.AlignLeftWithPanel	***True***
RelativePanel.AlignRightWithPanel	*False*
RelativePanel.AlignTopWithPanel	***True***
RelativePanel.AlignVerticalCenterWithPanel	*False*

HINWEIS: Die obigen Werte können Sie miteinander kombinieren, sodass sich alle neun Positionen innerhalb des *RelativPanels* zuweisen lassen.

Zusätzlich bieten sich die relativen Angaben bezogen auf ein weiteres Control an:

Eigenschaft	Bemerkung
Above	unter der Unterkante
LeftOf	an der linken Kante

Eigenschaft	Bemerkung
RightOf	an der rechten Kante
Below	auf der Oberkante
AlignBottomWith	gemeinsame Unterkante
AlignHorizontalCenterWith	gemeinsame vertikale Mittelachse
AlignLeftWith	gemeinsame linken Kante
AlignRightWith	gemeinsame rechten Kante
AlignTopWith	gemeinsame Oberkante
AlignVerticalCenterWith	gemeinsame horizontale Mittelachse

HINWEIS: Auch hier können Sie zwei Angaben miteinander kombinieren, um eine eindeutige Ausrichtung in x- und y-Richtung zu realisieren.

BEISPIEL 20.18: Beispiele für die Verwendung des *RelativPanel*

XAML

```
...
 <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
       Width="640" Height="300">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="319*"/>
            <ColumnDefinition Width="321*"/>
        </Grid.ColumnDefinitions>
```

Relativ zum Panel ausrichten (siehe untere Abbildung linke Hälfte):

```
        <RelativePanel Background="LightCyan">
            <Border Width="40" Height="40" Background="Yellow"
                    RelativePanel.AlignBottomWithPanel="True"/>
            <Border Width="40" Height="40" Background="Red"
                    RelativePanel.AlignLeftWithPanel="True"/>
            <Border Width="40" Height="40" Background="Blue"
                    RelativePanel.AlignVerticalCenterWithPanel="True"
                    RelativePanel.AlignRightWithPanel="True"/>
            <Border Width="40" Height="40" Background="Green"
                    RelativePanel.AlignHorizontalCenterWithPanel="True"
                    RelativePanel.AlignVerticalCenterWithPanel="True" />
        </RelativePanel>
```

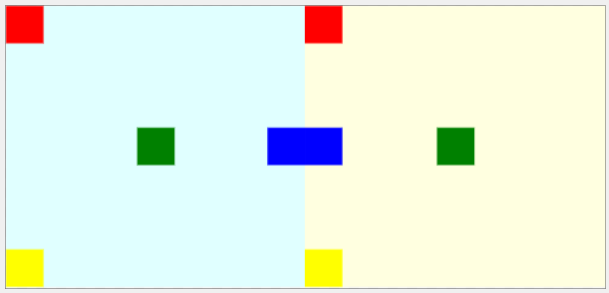
Relative Ausrichtung zu anderen Controls (siehe untere Abbildung rechte Hälfte):

```
        <RelativePanel Grid.Column="1" Background="LightYellow">
            <Border Name="linksunten" Width="40" Height="40" Background="Yellow"
                    RelativePanel.AlignBottomWithPanel="True"/>
            <Border Width="40" Height="40" Background="Red"
                    RelativePanel.AlignLeftWith="linksunten"/>
            <Border Name="mitte" Width="40" Height="40" Background="Green"
```

BEISPIEL 20.18: Beispiele für die Verwendung des *RelativPanel*

```
                    RelativePanel.AlignHorizontalCenterWithPanel="True"
                    RelativePanel.AlignVerticalCenterWithPanel="True" />
            <Border Width="40" Height="40" Background="Blue"
                    RelativePanel.AlignBottomWith="mitte"/>
        </RelativePanel>
    </Grid>
...
```

Ergebnis



20.3 Listendarstellungen

Im Zusammenhang mit der Anzeige von Datenlisten etc. werden auch entsprechende Controls benötigt, die diese Listen auf sinnvolle Weise visualisieren können.

20.3.1 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 20.19: 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>
...
```

BEISPIEL 20.19: Einfache *ListBox*, die Items werden per XAML definiert

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 20.20: Datenbindung *ListBox*

VB

```
...
Public NotInheritable Class Listen
    Inherits Common.LayoutAwarePage
```

Wir definieren die Liste als Eigenschaft der aktuellen Seite:

```
    Public Property Items() As List(Of String)
        Get
            Return m_Items
        End Get
        Set(value As List(Of String))
            m_Items = Value
        End Set
    End Property
    Private m_Items As List(Of String)
```

Im Konstruktor weisen wir der Liste einige Werte zu:

```
    Public Sub New()
         Me.InitializeComponent()
         Items = New List(Of String)()
         For i As Integer = 0 To 99
             Items.Add("Eintrag " + i.ToString())
         Next
     End Sub
 ...
```

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:

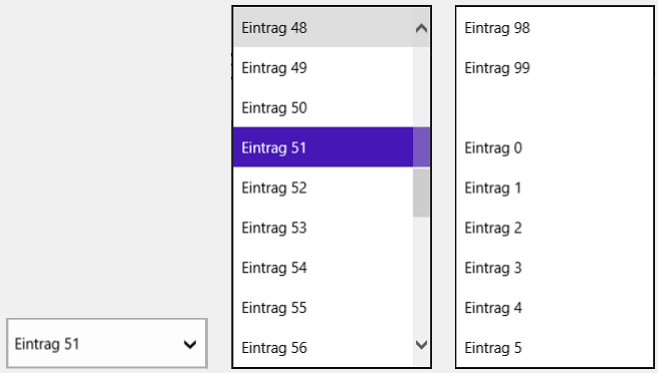
BEISPIEL 20.21: Verwendung *ComboBox*

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 20.22: Binden an eine *ObservableCollection*

VB

Zunächst die Klasse *Artikel* definieren (wir implementieren *INotifyPropertyChanged*, um auf Eigenschaftsänderungen reagieren zu können):

```
...
Public Class Artikel
    Implements INotifyPropertyChanged
```

Die beiden Eigenschaften Name und Preis:

```
    Private _name As String
    Public Property Name() As String
        Get
```

BEISPIEL 20.22: Binden an eine *ObservableCollection*

```
            Return _name
        End Get
        Set(value As String)
            _name = value
            Me.OnPropertyChanged("Name")
        End Set
    End Property

    Private _preis As Double

    Public Property Preis() As Double
        Get
            Return _preis
        End Get
        Set(value As Double)
            _preis = value
            Me.OnPropertyChanged("Preis")
        End Set
    End Property

    Public Sub New()
    End Sub
```

Die statische Methode *GetBeispielArtikel* liefert uns eine bereits gefüllte Liste von Artikeln:

```
   Public Shared Function GetBeispielArtikel() As ObservableCollection(Of Artikel)
        Dim _artikel As New ObservableCollection(Of Artikel)() From { _
            New Artikel() With { _
                .Name = "Bier", _
                .Preis = 1.2 _
            }, _
            New Artikel() With { _
                .Name = "Wein", _
                .Preis = 3.7 _
            }, _
...
            New Artikel() With { _
                .Name = "Kohlrabi", _
                .Preis = 0.5 _
            } _
        }
        Return _artikel
    End Function
```

Die Implementierung des *PropertyChanged*-Ereignisses:

```
   Public Event PropertyChanged(sender As Object, e As PropertyChangedEventArgs) _
               Implements INotifyPropertyChanged.PropertyChanged
```

BEISPIEL 20.22: Binden an eine *ObservableCollection*

```
    Protected Overridable Sub OnPropertyChanged(propertyName As String)
        RaiseEvent PropertyChanged(Me, New _
             System.ComponentModel.PropertyChangedEventArgs(propertyName))
    End Sub
...
```

In der eigentlichen Seite definieren wir zunächst eine Eigenschaft vom Typ *ObservableCollection<Artikel>* und weisen dieser die Beispieldaten zu:

```
...
    Public Property Artikeliste() As ObservableCollection(Of Artikel)
        Get
            Return m_Artikeliste
        End Get
        Set(value As ObservableCollection(Of Artikel))
            m_Artikeliste = value
        End Set
    End Property
    Private m_Artikeliste As ObservableCollection(Of Artikel)

    Public Sub New()
        Me.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

20.3.2 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 20.23: Verwendung *ListView*

XAML

```
...
    <ListView ItemsSource="{Binding Artikeliste, ElementName=pageRoot}" Width="150"
        Height="450" VerticalAlignment="Top" SelectionMode="Extended" Header="Artikel">
```

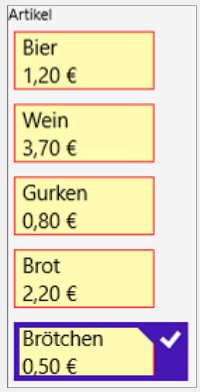
Wie Sie sehen, binden wir die *ListView* ebenfalls an die schon aus dem vorhergehenden Beispiel 20.19 bekannte Artikelliste.

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:

```
            <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 20.5.1 ab Seite 1015 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!

20.3.3 GridView

Möchten Sie die Daten nicht nur in einer Spalte sondern auch mehrspaltig anzeigen, können Sie ein *GridView* verwenden.

BEISPIEL 20.24: Verwendung *GridView*

XAML

Wir binden die *GridView* an die aus dem Beispiel 19.19 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="5,0,5,0" 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"
```

BEISPIEL 20.24: Verwendung *GridView*

```
                                VerticalChildrenAlignment="Top"
                                HorizontalChildrenAlignment="Left"/>
                        </ItemsPanelTemplate>
                    </GridView.ItemsPanel>
                </GridView>
            </StackPanel>
...
```

VB

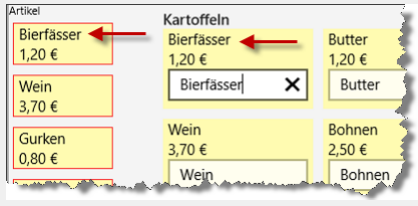
Über das *ItemClick*-Ereignis (dieses muss mit *IsItemClickEnabled* freigeschaltet werden) können wir den aktuell gewählten Item auswerten (Typisierung nicht vergessen):

```
...
    Private Sub GridView_ItemClick_1(sender As Object, e As ItemClickEventArgs)
        Dim item As Artikel = TryCast(e.ClickedItem, Artikel)
        txt1.Text = item.Name
    End Sub
...
```

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 20.5 ab Seite 1015.

20.3.4 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 20.25: 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 20.26: Verwendung *ListView* mit Datenbindung

XAML

```
...
            <FlipView Width="400" Height="100" ItemsSource="{Binding Artikeliste,
                      ElementName=pageRoot}" Background="AliceBlue">
```

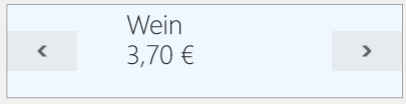
BEISPIEL 20.26: Verwendung *ListView* mit Datenbindung

```
                <FlipView.ItemTemplate>
                    <DataTemplate>
                        <StackPanel Margin="120,0,120,0">
                            <TextBlock Text="{Binding Name}" HorizontalAlignment="Left"
                                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 20.19 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>
```

20.4 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.

20.4.1 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 20.27: 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>
...
```

VB

```
...
Imports Windows.Media.Capture
Imports Windows.Devices.Enumeration
...
Public NotInheritable Class Capture_Beispiel
     Inherits Page
```

Eine Instanz der *MediaCapture*-Klasse definieren:

```
    Private medcap As MediaCapture
```

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 Sub pageRoot_Loaded(sender As Object, e As RoutedEventArgs)
         Dim videocams As DeviceInformationCollection = Await _
                   DeviceInformation.FindAllAsync(DeviceClass.VideoCapture)
```

Ist mindestens ein Videogerät vorhanden, wählen wir dieses aus und initialisieren sowohl die *MediaCapture*-Instanz als auch das *CaptureElement*:

```
        If videocams.Count > 0 Then
```

BEISPIEL 20.27: Verwendung *CaptureElement*

```
            medcap = New MediaCapture()
            Dim mcis As New MediaCaptureInitializationSettings()
            mcis.VideoDeviceId = videocams(0).Id
            Await medcap.InitializeAsync()
            CaptureElement1.Source = medcap
        End If
    End Sub
```

Zu diesem Zeitpunkt wird noch nichts angezeigt, dies wird erst mit dem Aufruf der *StartPreviewAsync*-Methode realisiert:

```
    Private Async Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Await medcap.StartPreviewAsync()
    End Sub
```

Stoppen der Anzeige:

```
    Private Async Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        Await medcap.StopPreviewAsync()
    End Sub
```

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 Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        medcap.SetPreviewRotation(VideoRotation.Clockwise180Degrees)
    End Sub

End Class
```

HINWEIS: Starten Sie die App 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.

20.4.2 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 20.28: Laden eines Videos aus dem aktuellen App-Unterverzeichnis *videos*

VB
```
...
    Private Async Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        Dim folder As StorageFolder = Await _
            Package.Current.InstalledLocation.GetFolderAsync("videos")
        Dim file As StorageFile = Await folder.GetFileAsync("video2.mts")
        Dim stream = Await file.OpenAsync(FileAccessMode.Read)
        MediaElement1.SetSource(stream, file.ContentType)
     End Sub
...
```

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 20.29: 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 20.30: 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

20.4.3 Frame

Im Abschnitt 19.1.2 (Die Page, der Frame und das Window) ab Seite 955 haben wir Ihnen bereits das organisatorische Grundkonzept der Apps aufgezeigt. Ausgangspunkt der App war das einzige Fenster der App, als dessen Inhalt ein zentraler Frame zugewiesen wurde. Alle Seiten Ihrer 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 19.2 ab Seite 961).

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 Kapitelbeispiel).

20.4.4 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 20.31: Anzeige Webseite in einer *WebView*

VB
```
...
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Dim uri As New Uri("http://www.doko-buch.de")
        WebView1.Navigate(uri)
    End Sub
...
```

BEISPIEL 20.32: Anzeige HTML-Code in einer WebView

VB
```
...
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        WebView1.NavigateToString("<html><body><b>" & _
               "Hier könnte Ihr HTML-Text stehen </b></body></html>")
    End Sub
...
```

Einziges verwertbares Feedback des Controls ist neben dem *NavigationFailed* das *LoadCompleted*-Ereignis:

BEISPIEL 20.33: Auswerten des *LoadCompleted*-Ereignisses

VB
```
...
    Private Sub pageRoot_Loaded_1(sender As Object, e As RoutedEventArgs)
        AddHandler WebView1.LoadCompleted, AddressOf WebView1_LoadCompleted
    End Sub
```

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:

```
    Private Sub WebView1_LoadCompleted(sender As Object, e As NavigationEventArgs)
        Dim b As New WebViewBrush()
        b.SourceName = "WebView1"
        b.Redraw()
        Rectangle1.Fill = b
    End Sub

...
```

20.4.5 ToolTip

Als kleine Hilfe für den Anwender stehen auch in 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 20.34: 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 20.34: Verwendung von *ToolTips*

Ergebnis

20.4.6 CalendarDatePicker

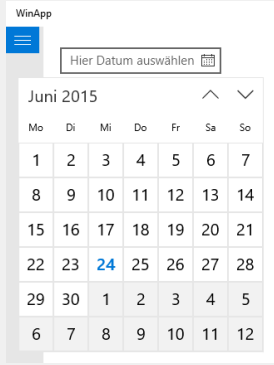
Ein häufig unentbehrliches Eingabe-Control gibt sich jetzt auch für die App-Entwicklung. Die Rede ist von einer Datumsauswahl, die vom *CalendarDatePicker* bereitgestellt wird. Die Verwendung ist trivial, ein kurzes Beispiel genügt:

BEISPIEL 20.35: *CalendarDatePicker*

XAML

```
...
 <StackPanel Margin="20">
       <CalendarDatePicker x:Name="Calendar1" Margin="0,0,0,20"
                           FirstDayOfWeek="Monday"
                           PlaceholderText="Hier Datum auswählen"
                           IsTodayHighlighted="True"
                           DateChanged="CalendarDatePicker_DateChanged" />
       <TextBlock Name="tb1"  Text="Noch kein  Datum gewählt!"/>
   </StackPanel>
...
```

Ergebnis



20.4.7 DatePicker/TimePicker

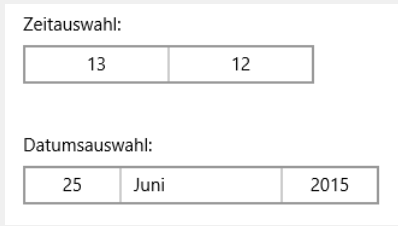
Schon seit Windows 8.1 dabei, wollen wir Ihnen die beiden Controls für die einfache Zeit- bzw. Datumsauswahl nicht vorenthalten.

BEISPIEL 20.36: ***DatePicker*** **und** ***TimePicker***

XAML

```
...
 <StackPanel Margin="20">
       <TimePicker Header="Zeitauswahl:" Margin="0,20,0,20" />
       <DatePicker Header="Datumsauswahl:" Margin="0,20,0,20" />
   </StackPanel>
...
```

Ergebnis



HINWEIS: Beide Controls sind für die Toucheingabe optimiert und zeigen nach der Auswahl entsprechende Auswahllisten für die gültigen Wertbereich an

20.5 Praxisbeispiele

20.5.1 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
     Implements IValueConverter

     Public Function Convert(value As Object, targetType As Type, parameter As Object,
          language As String) As Object Implements IValueConverter.Convert
         If parameter Is Nothing Then Return value
         Return String.Format(DirectCast(parameter, String), value)
     End Function
```

```
    Public Function ConvertBack(value As Object, targetType As Type, parameter As Object,
          language As String) As Object Implements IValueConverter.ConvertBack
        Return value
    End Function
End Class
```

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>
    <local: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 NotInheritable Class BasicPage1
    Inherits Common.LayoutAwarePage

    Public Property Datum() As DateTime
        Get
            Return m_Datum
```

```
        End Get
        Set(value As DateTime)
            m_Datum = Value
        End Set
    End Property

    Private m_Datum As DateTime
    Public Property Preis() As Double
        Get
            Return m_Preis
        End Get
        Set(value As Double)
            m_Preis = Value
        End Set
    End Property
    Private m_Preis As Double

    Public Sub New()
        Me.InitializeComponent()
        Datum = System.DateTime.Now.AddMonths(5)
        Preis = 123456.78
...
    End Sub
...
```

Test

Die Ausgabe:

```
14. November 2012
14.11.2012
123.456,78 €
1,234568E+005
123456,78
```

20.5.2 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 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 verwen-

den. Hier wird beim Fokuserhalt eine zusätzliche virtuelle Tastatur eingeblendet, die überhaupt erst eine Texteingabe ermöglicht:

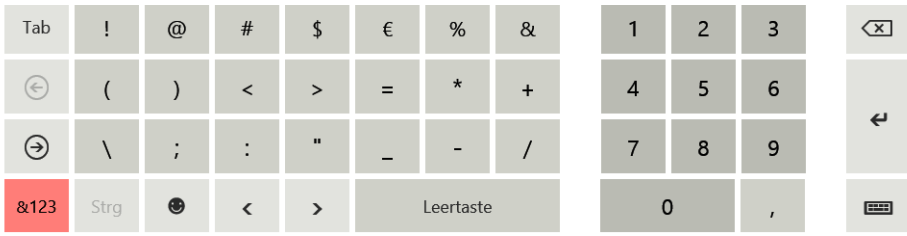
Je nach Einsatzzweck können Sie diese Tastatur auch anpassen. Nutzen Sie dafür die *InputScopeName*-Eigenschaft, die Sie allerdings nur recht umständlich setzen können:

```
<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 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 Sub TextBox_KeyDown_1(sender As Object, e As KeyRoutedEventArgs)
    e.Handled = (e.Key < Windows.System.VirtualKey.Number0 Or _
                 e.Key > Windows.System.VirtualKey.Number9)
End Sub
```

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:

```
...
    Private txt1value As String = ""
...
    Private Sub TextBox_GotFocus_1(sender As Object, e As RoutedEventArgs)
        txt1value = TryCast(sender, TextBox).Text
    End Sub
```

Ändert sich der Inhalt der *TextBox* (z.B. Einfügen der Zwischenablage), führen wir folgende Ereignisprozedur aus:

```
    Private Sub TextBox_TextChanged_1(sender As Object, e As TextChangedEventArgs)
```

Eine Fehlerbehandlung für den Ernstfall:

```
        Try
```

Alle Zeichen außer 0...9 herausfiltern:

```
            Dim regex As New Regex("[^0-9]")
            Dim newvalue As String = regex.Replace(TryCast(sender, TextBox).Text, String.Empty)
```

Sollte kein Wert mehr vorhanden sein, setzen wir den Wert auf Null:

```
            If newvalue = String.Empty Then newvalue = "0"
```

Zur Sicherheit testen wir die Umwandlung in *Int32*:

```
            Dim val As Int32 = Convert.ToInt32(newvalue)
```

Hat alles geklappt, übernehmen wir den Wert:

```
          txt1value = newvalue
          TryCast(sender, TextBox).Text = newvalue
        Catch generatedExceptionName As Exception
```

Im Fehlerfall stellen wir den ursprünglichen Wert wieder her[1]:

```
            TryCast(sender, TextBox).Text = txt1value
        End Try
    End Sub
```

Damit beschränkt sich die Eingabe in der *TextBox* auf gültige *Int32*-Zahlen.

20.5.3 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 grup-

[1] Eine Undo-Methode ist leider nicht vorhanden.

pieren. 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"
                             ItemsPath="Items" />
    </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
```

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>
```

[1] Den *StringFormatConverter* kennen Sie bereits aus dem vorhergehenden Beispiel.

```
                </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"
                                       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 20.19 in das Projekt und passen Sie gegebenenfalls den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

```
...
Public NotInheritable Class BasicPage1
     Inherits Common.LayoutAwarePage

    Public Sub New()
        Me.InitializeComponent()
```

```
        Artikeliste = Artikel.GetBeispielArtikel()
```

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:

```
        Dim result = From art In Artikel.GetBeispielArtikel()
                     Group art By Key = art.Name.Substring(0, 1)
                     Into Group Select New With {.Key = Key, .Items = Group}
```

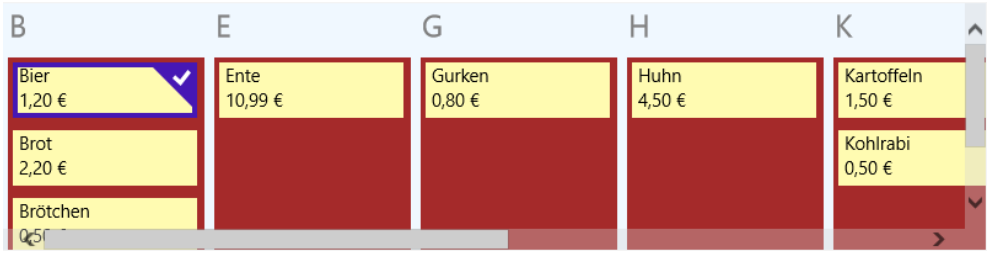
HINWEIS: Als Gruppierungsattribut verwenden wir den ersten Buchstaben des Artikelnamens. Sie könnten auch problemlos nach einem kompletten Feld oder auch nach unterschiedlichen Preisen gruppieren lassen.

```
        CollectionViewSource1.Source = result.OrderBy(Function(art) art.Key)
    End Sub
...
```

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>
```

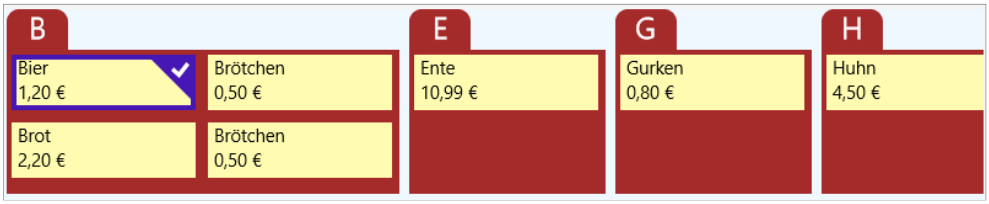
Wir ändern auch gleich noch das Aussehen des Gruppenkopfes mittels *Border*:

```
                            <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 1030 im Praxisbeispiel

► 20.5.5 Die CollectionViewSource verwenden.

20.5.4 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 20.5.3 gezeigt.

HINWEIS: Wir verwenden als Datenquelle die Artikelliste aus dem Beispiel 20.19.

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"
                              ItemsPath="Items" />
    </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 20.19 in das Projekt und passen Sie gegebenenfalls den Namespace an. Nachfolgend müssen wir uns nur noch um das Gruppieren der Daten und die Zuweisung der *Source* für die *CollectionViewSource* kümmern:

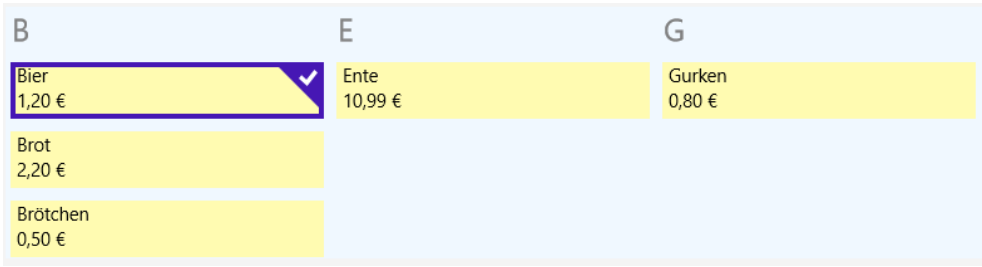
```
...
  Public Sub New()
        Me.InitializeComponent()
        Dim result = From art In Artikel.GetBeispielArtikel()
                     Group art By Key = art.Name.Substring(0, 1)
```

```
                    Into Group Select New With {.Key = Key, .Items = Group}
        CollectionViewSource1.Source = result.OrderBy(Function(art) art.Key)
    End Sub
```

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*:

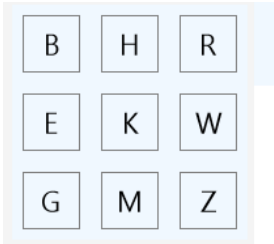


Die *ZoomOutView*:

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:



20.5.5 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 20.5.3 ab Seite 1020 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 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend können wir uns um den Konstruktor der Seite kümmern:

```
...
    Public Sub New()
        Me.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):

```
        AddHandler CollectionViewSource1.View.CurrentChanged, AddressOf View_CurrentChanged
        AddHandler CollectionViewSource1.View.CurrentChanging, AddressOf View_CurrentChanging
    End Sub
```

Wenn der aktuelle Artikel-Datensatz "Ente" ist, soll sich die Satzzeigerposition nicht mehr verändern lassen (die Änderung könnten Sie von weiteren Bedingungen abhängig machen):

```
    Private Sub View_CurrentChanging(sender As Object, e As CurrentChangingEventArgs)
        If TryCast(CollectionViewSource1.View.CurrentItem, Artikel).Name = "Ente" Then
            If e.IsCancelable Then
                e.Cancel = True
            End If
        End If
    End Sub
```

Hat sich die Satzzeigerposition erfolgreich geändert, zeigen wir den Namen des aktuell gewählten Artikels an:

```
    Private Sub View_CurrentChanged(sender As Object, e As Object)
        txt1.Text = TryCast(CollectionViewSource1.View.CurrentItem, Artikel).Name
    End Sub
```

Wechsel zum ersten Datensatz:

```
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        CollectionViewSource1.View.MoveCurrentToFirst()
    End Sub
```

Wechsel zum vorhergehenden Datensatz:

```
   Private Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        CollectionViewSource1.View.MoveCurrentToPrevious()
        If CollectionViewSource1.View.IsCurrentBeforeFirst Then
            CollectionViewSource1.View.MoveCurrentToFirst()
        End If
    End Sub
```

Wechsel zum nächsten Datensatz:

```
   Private Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        CollectionViewSource1.View.MoveCurrentToNext()
        If CollectionViewSource1.View.IsCurrentAfterLast Then
            CollectionViewSource1.View.MoveCurrentToLast()
        End If
    End Sub
```

Wechsel zum letzten Datensatz:

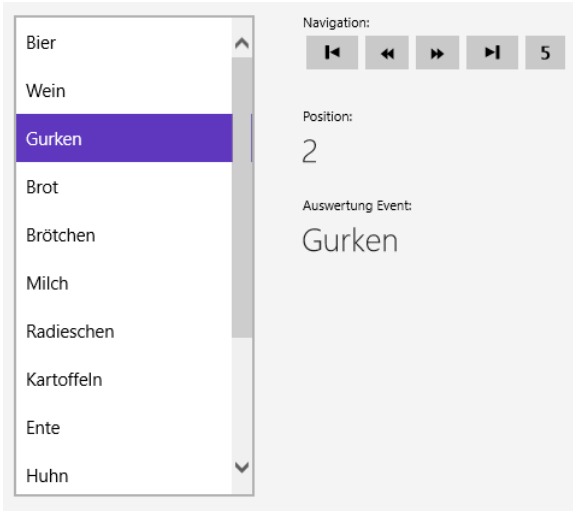
```
    Private Sub Button_Click_5(sender As Object, e As RoutedEventArgs)
        CollectionViewSource1.View.MoveCurrentToLast()
    End Sub
```

Sprung zum fünften Datensatz:

```
    Private Sub Button_Click_6(sender As Object, e As RoutedEventArgs)
        CollectionViewSource1.View.MoveCurrentToPosition(5)
    End Sub
...
```

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.



20.5.6 Zusammenspiel ListBox/AppBar

Es wird immer wieder propagiert: Windows App-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 20.19 in das Projekt und passen Sie den Namespace an. Nachfolgend kümmern wir uns um den Konstruktor der Seite:

```
...
Imports Windows.UI.Popups
...
Public NotInheritable Class BasicPage1
    Inherits Common.LayoutAwarePage

    Public Sub New()
        Me.InitializeComponent()
```

Daten laden:

```
        CollectionViewSource1.Source = Artikel.GetBeispielArtikel()
    End Sub
```

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 Sub ListView_SelectionChanged(sender As Object, e As SelectionChangedEventArgs)
        If Me.ListBox1.SelectedItems.Count > 0 Then
            Me.BottomAppBar1.IsSticky = True
            Me.BottomAppBar1.IsOpen = True
```

Keine Auswahl, keine Anzeige:

```
        Else
            Me.BottomAppBar1.IsOpen = False
            Me.BottomAppBar1.IsSticky = False
        End If
    End Sub
```

Klickt der Nutzer auf die *Delete*-Schaltfläche, zeigen wir eine Sicherheitsabfrage an:

```
Private Async Sub Delete_Click_1(sender As Object, e As RoutedEventArgs)
        Dim msg As String = "Datensatz wirklich löschen?"
        If ListBox1.SelectedItems.Count > 1 Then
            msg = "Datensätze wirklich löschen?"
```

MessageDialog erzeugen:

```
        Dim messageDialog = New MessageDialog(msg, "Sicherheitsabfrage")
```

Zwei Schaltflächen einblenden:

```
        messageDialog.Commands.Add(New UICommand("Ja",
             New UICommandInvokedHandler(AddressOf Me.CommandInvokedHandler), True))
        messageDialog.Commands.Add(New UICommand("Nein",
             New UICommandInvokedHandler(AddressOf Me.CommandInvokedHandler), False))
        messageDialog.DefaultCommandIndex = 1
        messageDialog.CancelCommandIndex = 1
```

Anzeige des Dialogs:

```
        Await messageDialog.ShowAsync()
    End Sub
```

Das Ergebnis des Dialogs können wir in folgendem Ereignis auswerten:

```
    Private Sub CommandInvokedHandler(command As IUICommand)
```

Wurde die *Ja*-Schaltfläche gedrückt, löschen wir alle markierten Einträge:

```
        Debug.WriteLine(command.Label + " " + command.Id.ToString())
        If CBool(command.Id) Then
            While ListBox1.SelectedItems.Count > 0
                TryCast(CollectionViewSource1.Source,
```

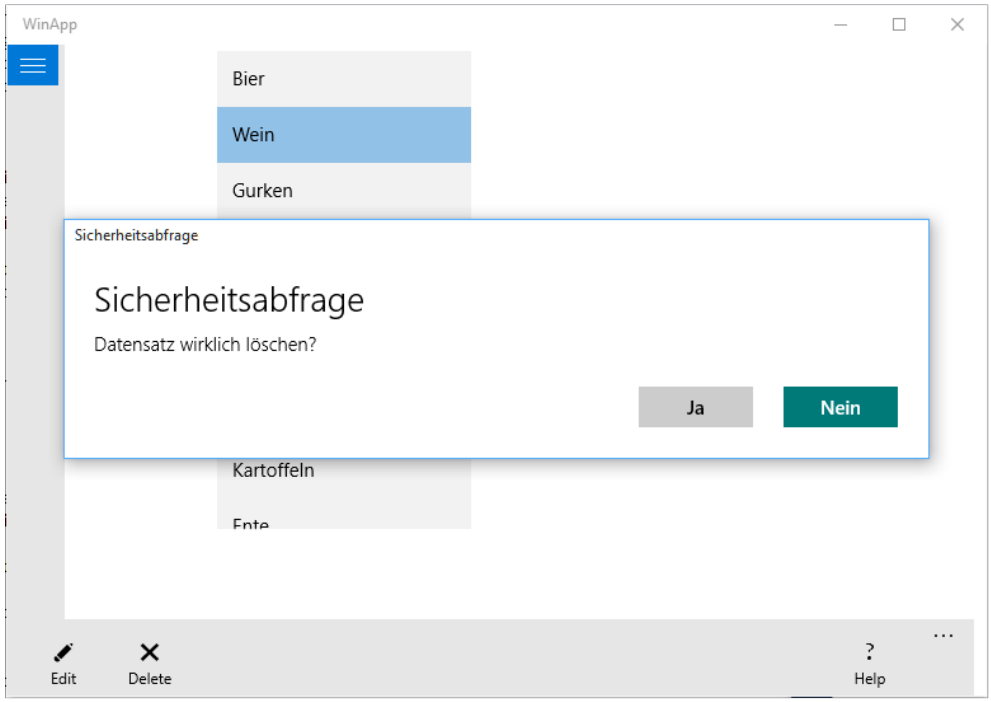
```
                    ObservableCollection(Of Artikel) _
                    ).Remove(TryCast(ListBox1.SelectedItems(0), Artikel))
            End While
        End If
    End Sub
```

Die Reaktion auf die *Edit*-Schaltfläche ist nur eine Statusmitteilung:

```
    Private Async Sub Edit_Click_1(sender As Object, e As RoutedEventArgs)
        Dim messageDialog = New MessageDialog("Funktion nicht implementiert!", "Hinweis")
        Await messageDialog.ShowAsync()
    End Sub
...
```

Test

Öffnen Sie die App, markieren Sie einige Einträge in der *ListBox* und versuchen Sie diese per *App-Bar* zu löschen:



Kapitel 21

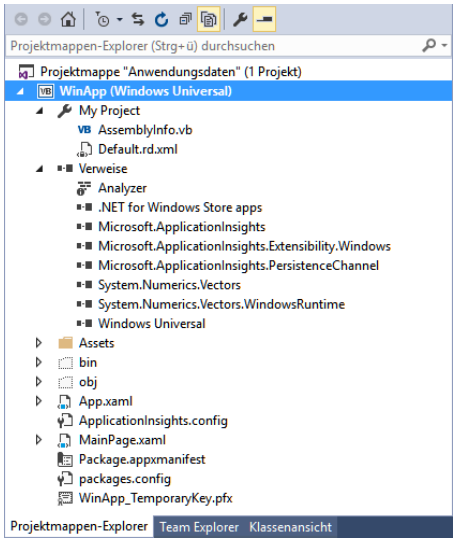
Apps im Detail

In diesem Kapitel wollen wir die Windows-Apps etwas 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.

21.1 Ein Windows App-Projekt im Detail

Sicher kann es nicht schaden, wenn wir uns zunächst etwas intensiver mit dem Grundgerüst einer Windows App beschäftigen, zumal der administrative Aufwand im Laufe der Jahre und mit jeder neuen Zielumgebung immer komplexer geworden ist. 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" Projekt erzeugt beim Einsteiger zunächst nichts als Ratlosigkeit, es sei denn, Sie haben sich vorher schon intensiv mit WPF oder Silverlight beschäftigt.



21.1.1 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 Windows Apps vertraut machen.

Im Zusammenhang mit 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 21.1.5, Seite 1041), so kann WinRT später die gewünschten Aufgaben anfordern (WinRT ist nur der Vermittler zwischen den einzelnen Vertragspartnern). Ein entsprechender Callback in der Datei *App.xaml.vb* ermöglicht Ihrem Programm die Reaktion auf die externen Anforderungen durch das System oder durch eine andere App[1].

- **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 21.4.1 ab Seite 1069.

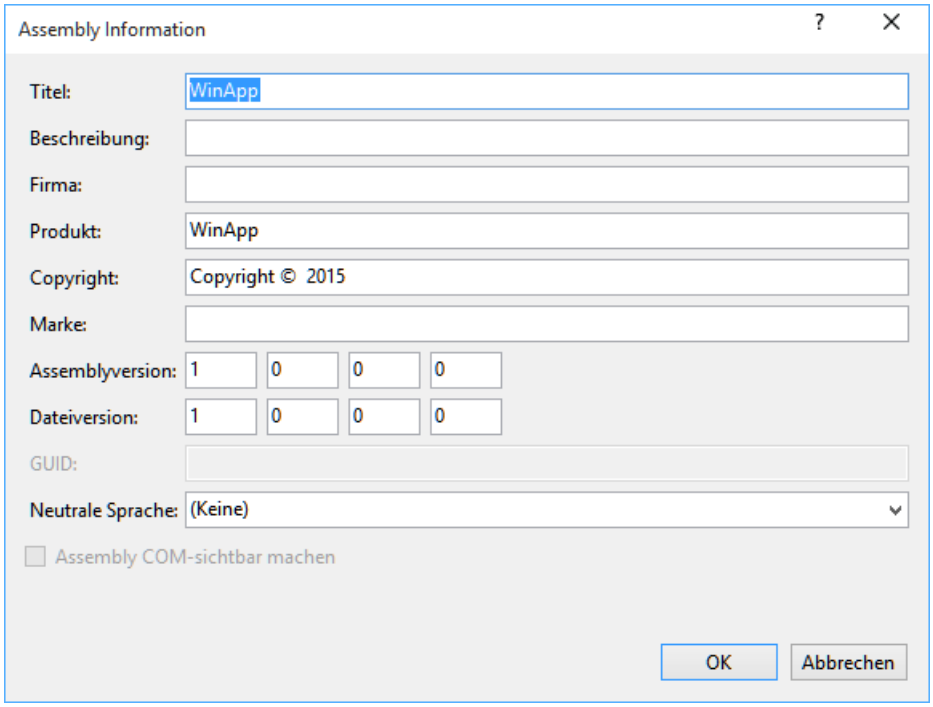
Nach diesem kurzen theoretischen Ausflug können wir uns nun endlich den einzelnen Dateien des App-Projekts widmen.

21.1.2 AssemblyInfo.vb

Diese Datei im Unterverzeichnis *Properties* speichert alle Informationen, die Sie auch über den Dialog *Projekt|Einstellungen|Anwendung|Assemblyinformationen* vornehmen können:

---

[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.vb* vor. Ohne diesen Contract startet keine App.



Die Daten werden intern als Attribute gespeichert und können wie folgt im laufenden Programm abgefragt werden:

BEISPIEL 21.1: Abfrage von Assemblyinformationen

VB

```
...
Imports System.Reflection
...
    Sub New()
        InitializeComponent()
```

Die einfache Variante funktioniert nur für einige Attribute (z.B. Versionsinformation):

```
        txt1.Text = GetType(MainPage).GetTypeInfo().Assembly.GetName().Version.ToString()
```

Mit der folgenden Variante können Sie alle gewünschten Attribute abrufen (jeweils eines):

```
        Dim txt = DirectCast(GetType(MainPage).GetTypeInfo().
                    Assembly.GetCustomAttribute(GetType(AssemblyDescriptionAttribute)),
                    AssemblyDescriptionAttribute)
          txt1.Text = txt.Description
     End Sub
...
```

HINWEIS: Wie Sie sehen, ist der Zugriff per Reflection auf die laufende Assembly in Apps neu organisiert worden. Die alte Version mit *System.Reflection.Assembly.GetExecutingAssembly()* funktioniert **nicht** mehr!

21.1.3 Verweise

Standardmäßig verweist Ihre App auf

- *.NET for Windows Store apps*
- *Microsoft.ApplicationInsights.xxx*
- *System.Numerics.Vectors.xx*
- *Windows Universal*

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\v4.5.1\
```

handelt, ist *Windows Universal* ein Verweis auf die Metadatendatei:

```
C:\Program Files (x86)\Windows Kits\10\References\Windows.Foundation.FoundationContract\1.0.0.0\
Windows.Foundation.FoundationContract.winmd
```

und damit auf alle *Windows.*-Namespaces, die Sie in Ihrer App benötigen.

Microsoft.ApplicationInsights bindet die Unterstützung für *Application Insights* in Ihr Projekt ein, ein entsprechender Aufruf findet sich in der *App.xaml.vb*-Datei. *Application Insights* bietet Ihnen die Möglichkeit, Nutzungs- und Performance-Analysen Ihrer Apps zu erstellen.

Für die Unterstützung der Hardware-Beschleunigung wird *System.Numerics.Vectors* eingebunden.

21.1.4 App.xaml und App.xaml.vb

Hierbei handelt es sich um den Ausgangs- und den Endpunkt für jede App. So ist in der Datei *App.xaml.vb* 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 19.1.2 "Die Page, der Frame und das Window" ab Seite 955 beschrieben. Mehr zu den einzelnen Methoden und Ereignissen dieser Klasse finden Sie im Abschnitt 21.2 "Der Lebenszyklus einer Windows App" ab Seite 1047.

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 21.2: 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

```
Dim  kundenDataSource As KundenDataSource = CType( _
                                App.Current.Resources("kundenDataSource"),  (KundenDataSource))
```

möglich. Da ist ein schlankes

```
App.KundenDataSource
```

wohl wesentlich einfacher.

21.1.5 Package.appxmanifest

Hierbei handelt es sich um das Manifest Ihrer App. Dieses dient dem WinRT-System als Informationsquelle über die Funktionalität Ihrer App.

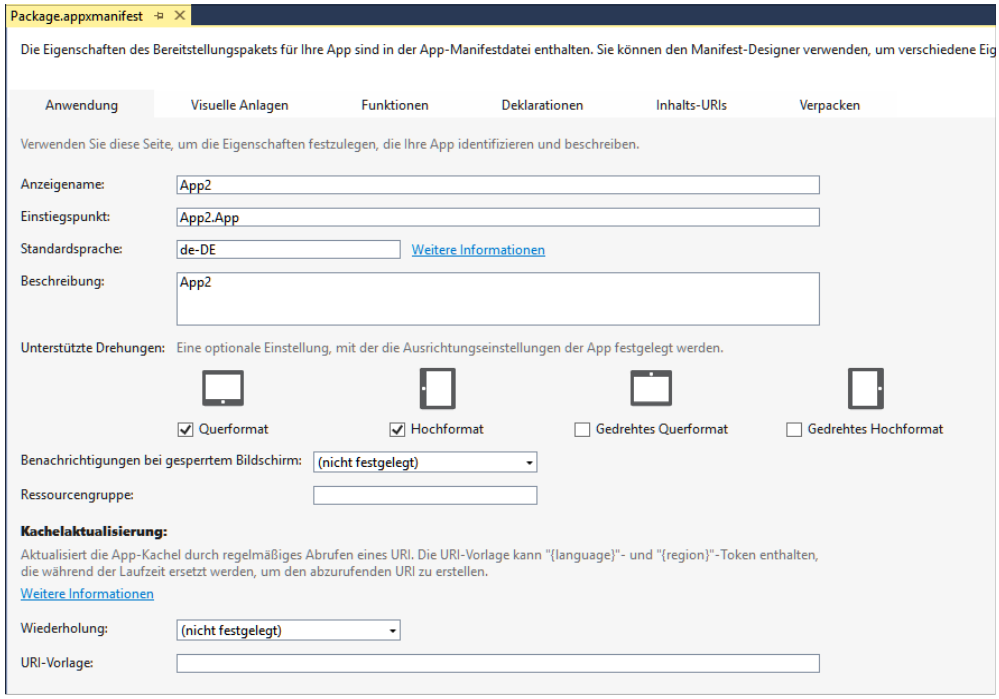
HINWEIS: Bevor Sie verzweifeln: Es handelt sich zwar um eine XML-Datei, Sie können jedoch einen recht komfortablen Editor für das Konfigurieren aller Einstellungen nutzen.

Der Editor ist in folgende vier Seiten eingeteilt:

- Anwendung
- Visuelle Anlagen
- Funktionen
- Deklarationen
- Inhalts-URIs
- Verpacken

Anwendung

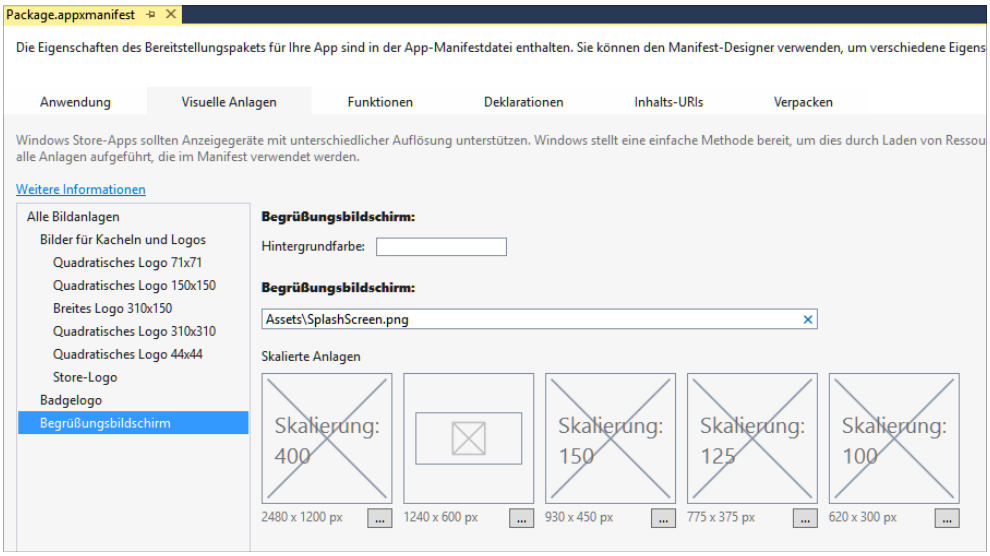
Diese Seite dürfte wohl für die wenigsten Fragen sorgen. So legen Sie zunächst den Anzeigenamen (dieser erscheint in der Übersicht aller Apps) und den Einstiegspunkt in Ihre App fest (Letzteren werden Sie wohl nie ändern).



Die Angaben zu den unterstützten Drehungen sind optional. Weiterhin finden Sie hier die Optionen für Benachrichtigungen und Kachelaktualisierungen.

Visuelle Anlagen

Legen Sie hier das Logo (Symbol) in verschiedenen Formaten, sowie die Schrift- und die Hintergrundfarbe Ihrer App im Startmenü fest.

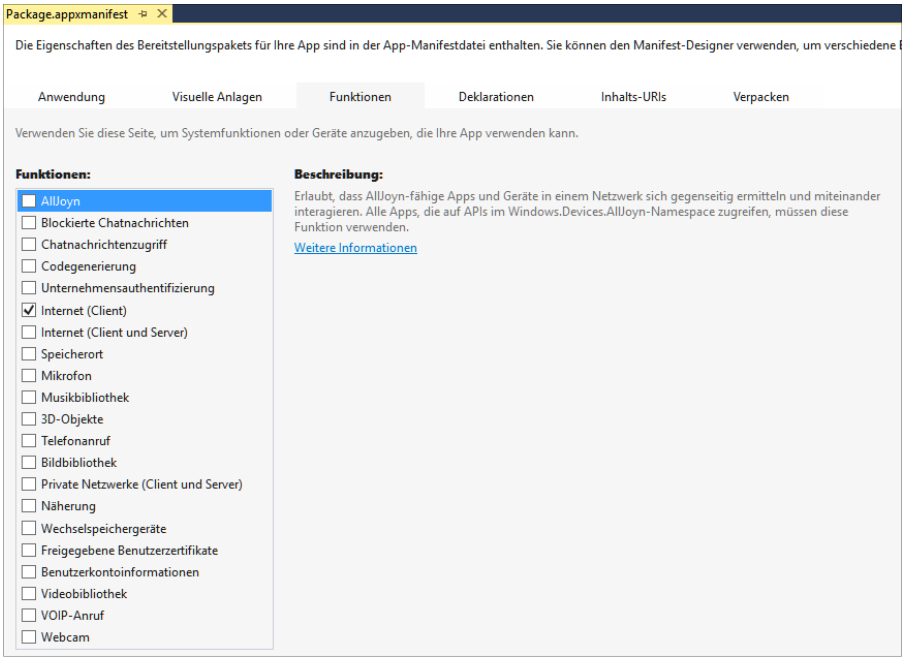


Last but not least, lassen sich auch das Logo sowie die Hintergrundfarbe des Splash Screens (Begrüßungsbildschirm) zuweisen.

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:

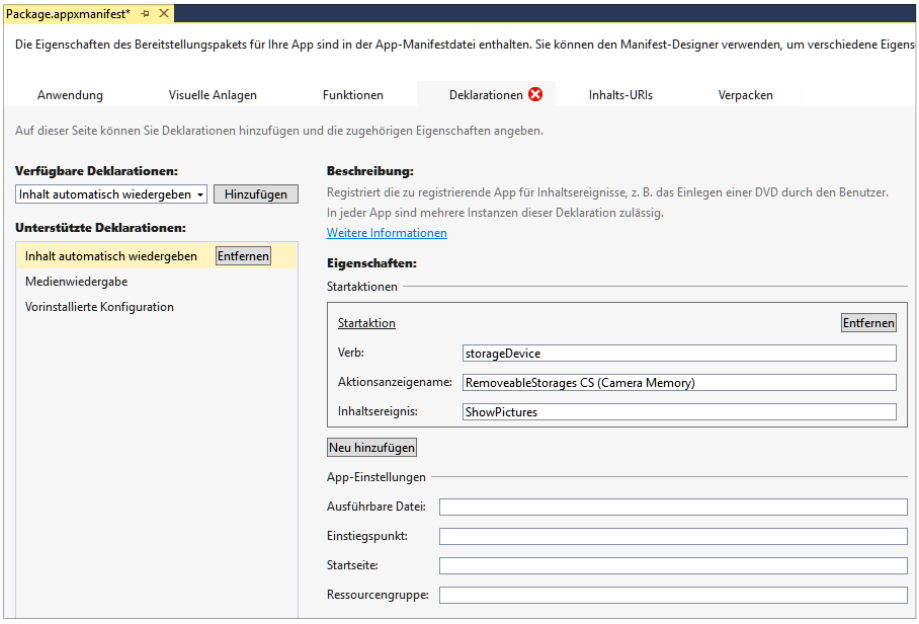
Funktion	Beschreibung
AllJoyn	AllJoyn-fähige Apps und Geräte können sich in einem Netzwerk gegenseitig erkennen und miteinander interagieren
Blockierte Chatnachrichten	Lesen blockierter SMS- und MMS-Nachrichten.
Chatnachrichtenzugriff	Lesen und Schreiben von Textnachrichten.
Codegenerierung	Ermöglicht der App das dynamische Generieren von Code
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

Funktion	Beschreibung
Näherung	Unterstützung für die neue Near-Field Communikation (NFC), die in Smartphones und Tablets zum Einsatz kommen soll.
Speicherort	Ermöglicht Ihrer App den Zugriff auf den aktuellen Standort des Tablets/Computers. Diese Information ist zum einen recht sensibel, zum anderen bietet sich hier aber auch die Möglichkeit, standortspezifische Programmfunktionen auszuführen.
Telefonanruf	Zugriff auf die Telefonfunktionen des Geräts freigeben.
VOIP-Anruf	Zugriff auf die VOIP-Anruf-APIs freigeben.
Benutzerkontoinformation	Zugriff auf Name und Bild des Benutzers ermöglichen.
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 folgenden 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, wann das Ereignis ausgelöst wird, der Anzeigename erscheint beim Registrieren der App für die automatische Wiedergabe.

Verpacken

Legen Sie hier die Optionen zum Verpacken der App fest. Dies betrifft vor allem den Anzeigenamen und das Zertifikat des Herausgebers:

21.1.6 Application1_TemporaryKey.pfx

Hierbei handelt es sich um ein selbst generiertes Zertifikat zum Signieren des App-Package. Jede Windows App 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.

21.1.7 MainPage.xaml & MainPage.xaml.vb

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.vb* fest. Siehe dazu auch

► 19.1.2 "Die Page, der Frame und das Window"

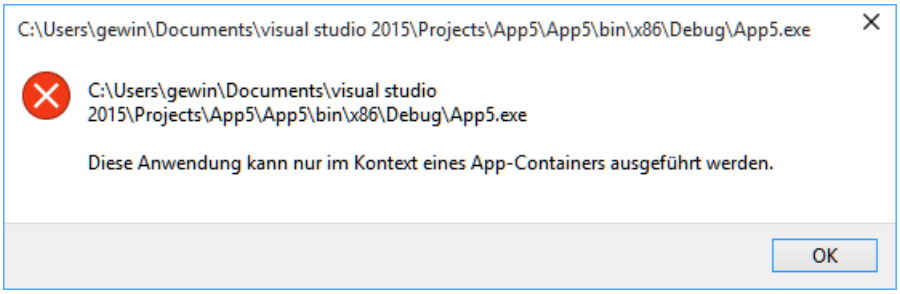
► 19.2 "Seitenauswahl und -navigation"

21.1.8 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 1041) 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, Store Logo: 50x50).

21.1.9 Nach dem Kompilieren

Haben Sie Ihre App kompiliert, sollten Sie auch einen Blick in das Projekt-Ausgabeverzeichnis *\\bin\\xxx\Debug* werfen. 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 Windows Apps zu finden. Diese erfolgt im Normalfall über den Windows Store. Von dort heruntergeladene Apps bzw. App-Packages lassen sich dann in Windows 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|App-Pakete erstellen* (siehe Abschnitt 22.5 ab Seite 1145).

21.2 Der Lebenszyklus einer Windows App

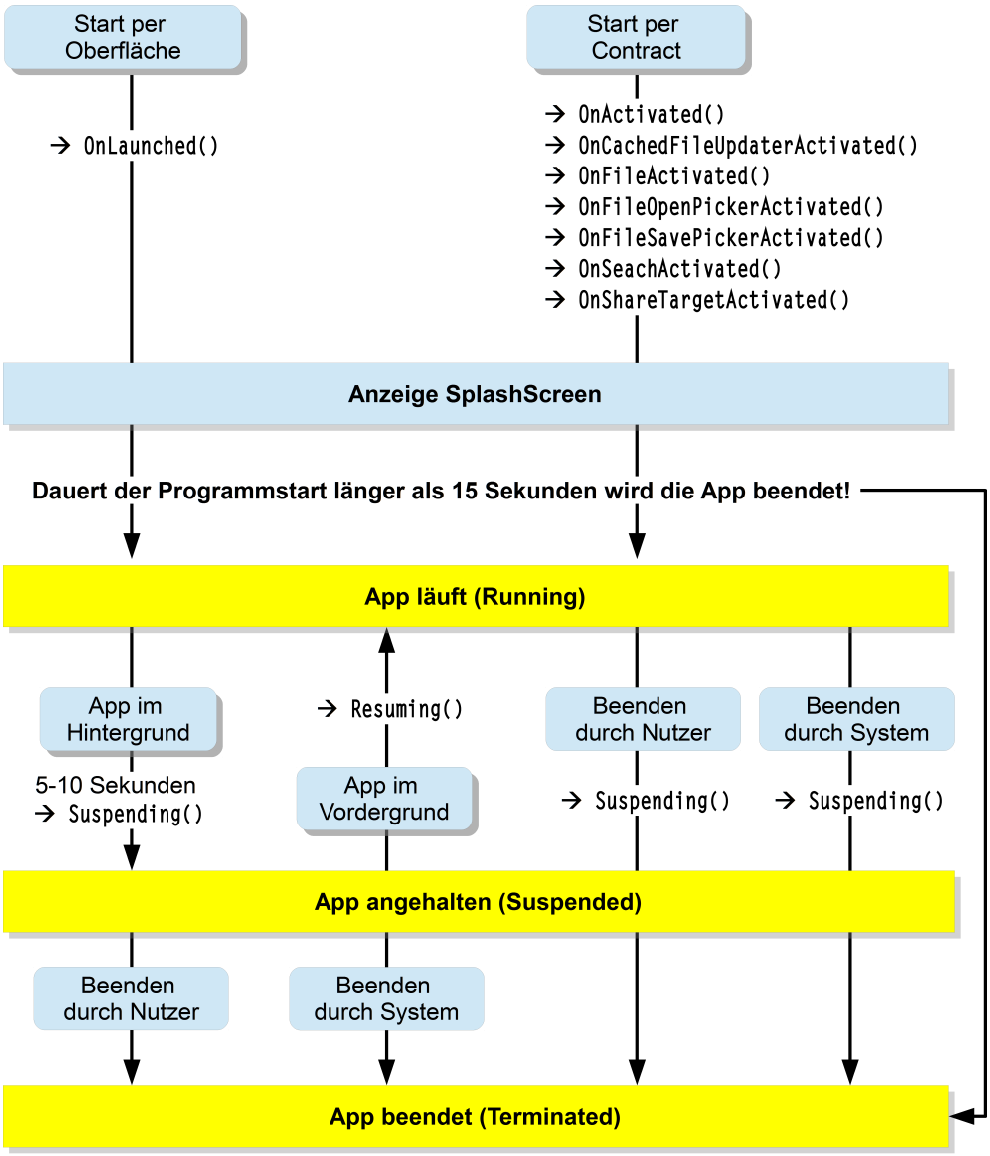
Wir hatten uns mit diesem Thema bereits an der einen oder anderen Stelle in den vorhergehenden Kapiteln kurz beschäftigt, hier soll es deshalb um einen zusammenfassenden Überblick gehen.

Wie schon erwähnt, werden für eine 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.

21.2.1 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 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.vb*) angelegten *OnLaunched*-Methode zu tun, die leider etwas verwirrend ist, dafür aber Raum für Anpassungen lässt:

```
    Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
```

Versuch, den Frame des App-Windows zu ermitteln:

```
        Dim rootFrame As Frame = Window.Current.Content
```

Ist dieser nicht bereits vorhanden, wird er erstellt:

```
       If rootFrame Is Nothing Then
            rootFrame = New Frame()
```

Die App wurde vorher vom System beendet:

```
            If args.PreviousExecutionState = ApplicationExecutionState.Terminated Then
                ' TODO: Zustand von zuvor angehaltener Anwendung laden
            End If
```

Rahmen platzieren:

```
            Window.Current.Content = rootFrame
        End If
```

Ist keine Seite geladen:

```
        If rootFrame.Content Is Nothing Then
```

Aufruf der Hauptseite mit Übergabe der Parameter:

```
            If Not rootFrame.Navigate(GetType(MainPage), args.Arguments) Then
```

```
                Throw New Exception("Failed to create initial page")
            End If
        End If
```

App-Window aktivieren:

```
        Window.Current.Activate()
    End Sub
```

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, 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. Siehe dazu auch

► 21.4.1 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.

21.2.2 Der Splash Screen

Haben Sie sich die Abbildung zum Lebenszyklus einer App auf Seite 1048 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 das genau funktioniert, zeigt das folgende Praxisbeispiel:

► 21.4.2 Einen zusätzlichen Splash Screen einsetzen

21.2.3 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

[1] Hintergrundprozesse werden natürlich noch weiter ablaufen.

transparent, stellen Sie sich also darauf ein, dass Ihr Programm aus dem Suspended-Mode nicht mehr "aufwacht"[1].

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 App vordefiniert ist.

```
Private Sub OnSuspending(sender As Object, e As SuspendingEventArgs) Handles Me.Suspending
    Dim deferral As SuspendingDeferral = e.SuspendingOperation.GetDeferral()
    ' TODO: Anwendungszustand speichern und alle Hintergrundaktivitäten beenden
    deferral.Complete()
End Sub
```

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*. Doch was ist wenn Sie umfangreichere Daten sichern wollen? Hier hilft nur konsequentes Umdenken beim Programmablauf. Sichern Sie Daten nicht erst zu diesem Zeitpunkt, d h. in letzter Sekunden, sondern dann wenn Sie anfallen (Eingaben, Seitenwechsel etc.).

Was sollten Sie eigentlich 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. In idesem Fall sollten Sie nur standardmäßige Programmeinstellungen laden

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.

21.2.4 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:

[1] Quasi gestorben im Schlaf ...

```
...
NotInheritable Class App
    Inherits Application

    Sub New()
        InitializeComponent()
        AddHandler Me.Resuming, AddressOf App_Resuming
    End Sub
...
    Private Sub App_Resuming(sender As Object, e As Object)
       Protokoll("OnResuming")
    End Sub

End Class
```

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.

21.2.5 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 21.3: App per *Button* schließen (Achtung: nicht regelkonform!)

VB
```
...
    Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Application.Current.Exit()
```

[1] Ja, auch das soll es auch noch geben!

BEISPIEL 21.3: App per *Button* schließen (Achtung: nicht regelkonform!)

```
    End Sub
...
```

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 1048), Sie können also Ihre Daten auch in diesem Fall problemlos sichern.

21.2.6 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.

21.2.7 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.

HINWEIS: Wer jetzt an *EventLog* oder *TraceListener* denkt, liegt falsch. Beides ist in Apps nicht verfügbar!

Das folgenden Beispiel zeigt eine einfache Lösung.

BEISPIEL 21.4: Möglichkeit zum Speichern einer Protokolldatei

VB

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:

```
...
Imports Windows.Storage
Imports System.Diagnostics
Imports System.Threading.Tasks
...
NotInheritable Class App
    Inherits Application
```

Die Datei, in der wir die Daten speichern:

```
    Private Shared _file As StorageFile
```

Ein Zwischenspeicher für die Protokolldaten:

```
    Private Shared _protokoll As New List(Of String)()
...
```

Zum Füllen der Protokollliste nutzen wir folgende Funktion:

```
    Private Shared Sub Protokoll(msg As String)
        Debug.WriteLine(msg)
        _protokoll.Add(Convert.ToString(DateTime.Now.ToString() + ": ") & msg)
    End Sub
```

Das eigentliche Sichern der Protokolldaten:

```
    Private Shared Async Sub SaveProtokoll()
        _file = Await KnownFolders.DocumentsLibrary.CreateFileAsync("MeinLogfile.txt",
                                            CreationCollisionOption.OpenIfExists)
        Await FileIO.AppendLinesAsync(_file, _protokoll)
        _protokoll.Clear()
    End Sub
```

Mit dem folgenden Aufruf wird das Protokoll gesichert:

```
    Private Async Sub OnSuspending(sender As Object, e As SuspendingEventArgs) _
                                        Handles Me.Suspending
        Dim deferral = e.SuspendingOperation.GetDeferral()
```

BEISPIEL 21.4: Möglichkeit zum Speichern einer Protokolldatei

```
        Protokoll("OnSuspending")
        SaveProtokoll()
        deferral.Complete()
    End Sub
...
```

Ergebnis

```
09.08.2015 09:57:42: OnLaunched Kind=Launch  PreviousExecutionState=ClosedByUser
09.08.2015 09:57:42: Window.Current.Activated
09.08.2015 09:57:52: Window.Current.Activated
09.08.2015 09:57:52: OnSearchActivated Kind=Search  PreviousExecutionState=Running
09.08.2015 09:57:56: Window.Current.Activated
09.08.2015 09:57:57: Window.Current.VisibilityChanged
09.08.2015 09:58:07: OnSuspending
09.08.2015 10:55:05: OnLaunched Kind=Launch  PreviousExecutionState=NotRunning
09.08.2015 10:55:06: Window.Current.Activated
09.08.2015 10:55:07: Window.Current.VisibilityChanged
09.08.2015 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

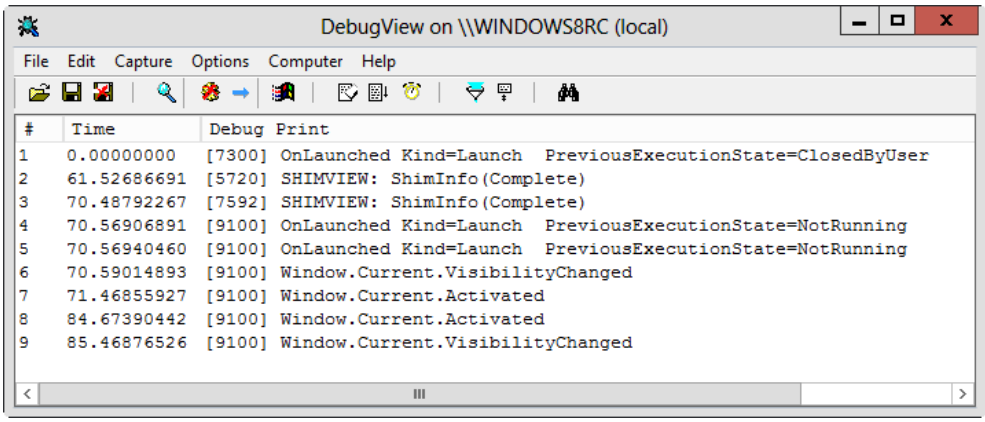
Sind Sie es leid, Ihre App unter den Einschränkungen des Visual Studio-Debuggers zu testen, nutzen Sie doch einfach *DebugView*, das in seiner aktuellen Version auch in der Lage ist, Debug-Messages (*Debug.WriteLine*) abzufangen und anzuzeigen.

Herunterladen können Sie das Tool unter der folgenden Adresse:

LINK: `http://technet.microsoft.com/en-us/sysinternals/bb896647`

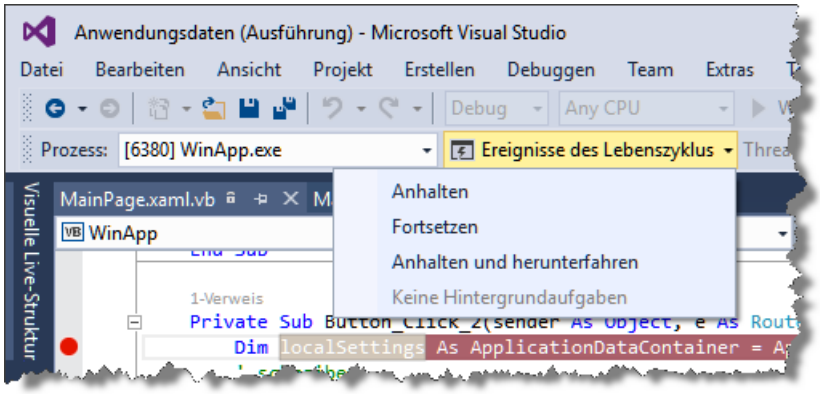
Entpacken Sie die enthaltenen Daten und starten Sie das Programm. Verteilen Sie in Ihrer App an den gewünschten Stellen die Debug-Aufrufe und starten Sie die App zunächst einmal aus Visual Studio heraus, um diese zu installieren. Beenden Sie dann die App und starten Sie aus der Windows-Oberfläche heraus.

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:

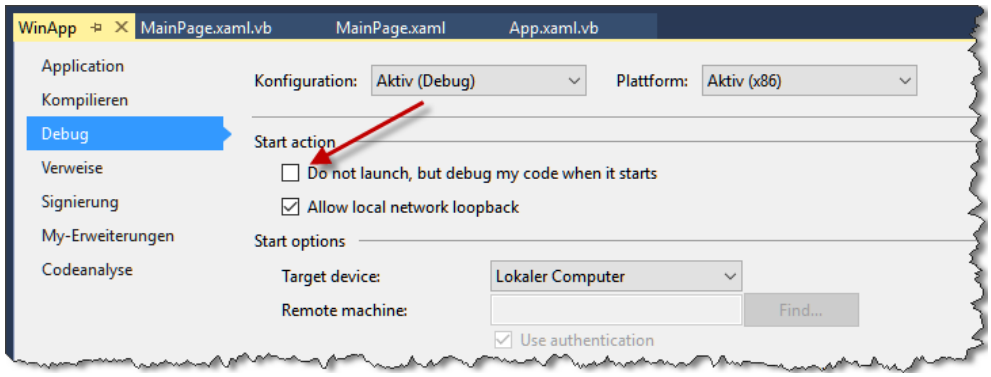


HINWEIS: Doch Achtung: Sie erhalten im Taskmanager kein Feedback, dass die App wirklich im Suspended-Modus ist. Das Ereignis *Suspending* wird hingegen ausgelöst.

Debuggen von Contract-aktivierten Apps

Soll Ihre App einen Contract unterstützt 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 implemen-

tiert. 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 Apps ab und wenden uns einem neuen Thema zu.

21.3 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 kennengelernt, nun wollen wir das Ganze noch einmal zusammenhängend betrachten.

21.3.1 Grundsätzliche Überlegungen

Bevor Sie jetzt an wohlvertraute und bekannten Konzepte denken, sollten Sie zunächst überlegen, ob sich 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.

21.3.2 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 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

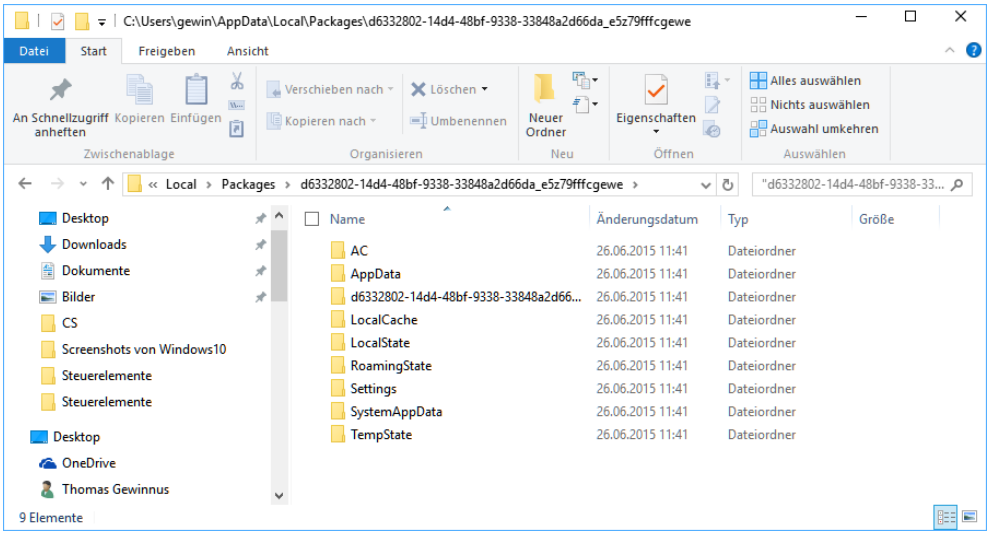
Sehen wir uns die einzelnen Verzeichnisse einmal im Detail an.

21.3.3 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 21.5: Erzeugen einer Datei im Unterordner *LocalState*

VB
```
...
Imports Windows.Storage
...
    Private Async Sub Button_Click_7(sender As Object, e As RoutedEventArgs)
        Dim localFolder = ApplicationData.Current.LocalFolder
        Dim file = Await localFolder.CreateFileAsync("Testdatei.txt")
    End Sub
...
```

Doch Vorsicht:

HINWEIS: Daten, die Sie hier ablegen, sind zwar vor dem Zugriff durch andere 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 21.6: 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 21.7: Erzeugen einer Roaming-Datei im Unterordner *RoamingState*

VB
```
...
  Private Sub Button_Click_8(sender As Object, e As RoutedEventArgs)
    Dim roamingFolder = ApplicationData.Current.RoamingFolder
    Dim file = Await roamingFolder.CreateFileAsync("Testdatei.txt",
```

BEISPIEL 21.7: Erzeugen einer Roaming-Datei im Unterordner *RoamingState*

```
                              CreationCollisionOption.ReplaceExisting)
    ApplicationData.Current.SignalDataChanged()
  End Sub
...
```

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.

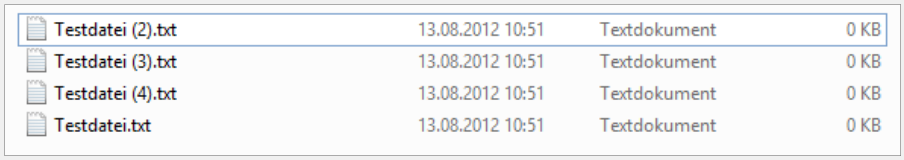
BEISPIEL 21.8: Erzeugen einer temporären Datei im Unterordner *TempState*

VB

```
...
  Private Sub Button_Click_8(sender As Object, e As RoutedEventArgs)
    Dim tempFolder = ApplicationData.Current.TemporaryFolder
    Dim file = Await tempFolder.CreateFileAsync("Testdatei.txt",
                                                CreationCollisionOption.GenerateUniqueName)
  End Sub
...
```

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"/>
```

21.3.4 Das Anwendungs-Installationsverzeichnis

Neben dem App-Datenverzeichnis besitzt jede App auch ein spezifisches Installationsverzeichnis, das Sie unter folgendem Pfad finden[1]:

[1] Dies gilt nur bei einer ordnungsgemäßen Installation per Store oder PowerShell-Skript.

```
C:\Program Files\WindowsApps\<Package-ID>
```

Hier haben Sie lesenden Zugriff auf alle Dateien Ihres Installations-Packages.

BEISPIEL 21.9: Inhalte des Installationsverzeichnisses auflisten

VB
```
...
  Private Sub Button_Click_9(sender As Object, e As RoutedEventArgs)
    Dim installlocFolder As StorageFolder = _
                        Windows.ApplicationModel.Package.Current.InstalledLocation
    ListBox1.Items.Add(installlocFolder.Path)
    For Each file In Await installlocFolder.GetFilesAsync()
      ListBox1.Items.Add("    -->" + file.Path)
    Next
    For Each folder In Await installlocFolder.GetFoldersAsync()
      ListBox1.Items.Add(folder.Path)
      For Each file In Await folder.GetFilesAsync()
        ListBox1.Items.Add("    -->" + file.Path)
      Next
    Next
  End Sub
...
```

Der Zugriff per XAML:

```
<Image Source="ms-appx:///myFile.png"/>
```

21.3.5 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:

```
...
Dim folder = Await DownloadsFolder.CreateFolderAsync("Testen")
...
```

Versuchen Sie aber dieses Verzeichnis später erneut zu öffnen

```
Dim 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 21.10: Datei im Downloads-Verzeichnis anlegen

VB
```
...
Imports Windows.Storage
...
  Private Sub Button_Click_10(sender As Object, e As RoutedEventArgs)
```

Unterordner erzeugen:

```
    Dim folder = Await DownloadsFolder.CreateFolderAsync("Testen",
                                       CreationCollisionOption.GenerateUniqueName)
```

Datei in den Unterordner schreiben:

```
    Dim file = Await folder.CreateFileAsync("Testdatei.txt",
                                          CreationCollisionOption.GenerateUniqueName)
```

Datei direkt in *Downloads* schreiben:

```
    Dim file2 = Await DownloadsFolder.CreateFileAsync("Testdatei.txt",
                                          CreationCollisionOption.GenerateUniqueName)
  End Sub
...
```

21.3.6 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 21.1.1 ab Seite 1038 und Abschnitt 21.1.5 Seite 1041) 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 21.11: Alle Bilder in *Bibliotheken|Bilder* auflisten

VB
```
...
     Private Sub Button_Click_11(sender As Object, e As RoutedEventArgs)
        Dim pictures As StorageFolder = KnownFolders.PicturesLibrary
        For Each file In Await pictures.GetFilesAsync()
```

BEISPIEL 21.11: Alle Bilder in *Bibliotheken|Bilder* auflisten

```
            ListBox1.Items.Add("    -->" + file.Path)
        Next
    End Sub
...
```

HINWEIS: Vergessen Sie nicht die entsprechenden Funktionen freizuschalten, ansonsten wird der Zugriff blockiert!

21.3.7 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 21.3.3 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 OneDrive) gespeichert werden.

Schauen wir uns also an, wie verschiedene Arten von Daten mit Hilfe der Anwendungseinstellungen gesichert werden können.

BEISPIEL 21.12: Einfache Werte sichern und laden

VB

```
...
Imports Windows.Storage
...
        Dim localSettings As ApplicationDataContainer =
                    ApplicationData.Current.LocalSettings
```

Einen Wert schreiben:

```
        localSettings.Values("EinfacherWert") = "Ein gespeicherter Wert!"
```

Den Wert später wieder lesen:

```
        ListBox1.Items.Add(DirectCast(localSettings.Values("EinfacherWert"), String))
...
```

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 21.13: Zusammengesetzte Werte speichern

VB
```
...
        Dim localSettings As ApplicationDataContainer = _
                                            ApplicationData.Current.LocalSettings
        Dim composite As New ApplicationDataCompositeValue()
        composite("Username") = "Müller"
        composite("Punktestand") = CInt(1000)
```

Zusammengesetzten Wert speichern:

```
        localSettings.Values("BeispielCompositeValue") = composite
```

Zusammengesetzten Wert laden:

```
        composite = DirectCast(localSettings.Values("BeispielCompositeValue"),
                                              ApplicationDataCompositeValue)
        ListBox1.Items.Add(DirectCast(composite("Username"), String))
        ListBox1.Items.Add(CInt(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 21.14: Werte in Containern speichern

VB
```
...
        Dim localSettings As ApplicationDataContainer =
                                          ApplicationData.Current.LocalSettings
```

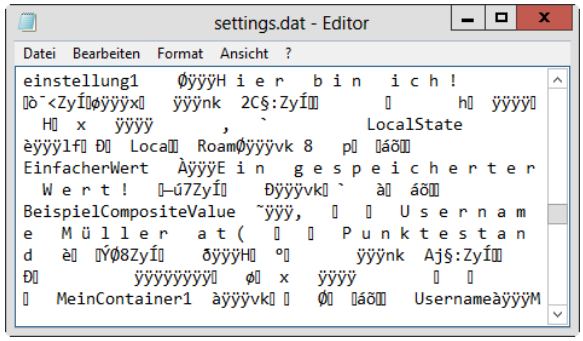
Container erstellen:

```
        Dim container As ApplicationDataContainer =
                                localSettings.CreateContainer("MeinContainer1",
                                ApplicationDataCreateDisposition.Always)
        If localSettings.Containers.ContainsKey("MeinContainer1") Then
```

Daten in den Container schreiben:

```
           localSettings.Containers("MeinContainer1").Values("Username") = "Müller"
        End If
     End Sub
...
```

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:



21.3.8 Daten in der Cloud ablegen/laden (Roaming)

Wie schon erwähnt, bieten Windows 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 1060) 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 21.15: Speichern und Laden eines Wertes per Roaming

VB

```
        Dim roaming As ApplicationDataContainer = ApplicationData.Current.RoamingSettings
        roaming.Values("einstellung1") = "Gesichert am " + DateTime.Now.ToString()
        ApplicationData.Current.SignalDataChanged()
...
        Dim roaming As ApplicationDataContainer = ApplicationData.Current.RoamingSettings
        If roaming.Values("einstellung1") IsNot Nothing Then
           ListBox1.Items.Add(DirectCast(roaming.Values("einstellung1"), String))
        End If
```

Das gültige Maximum für die synchronisierbare Datenmenge erfahren Sie über *ApplicationData.Current.RoamingStorageQuota*.

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 21.16: Reaktion auf das *DataChanged*-Ereignis

VB

```
...
      Public Sub New()
         Me.InitializeComponent()
         AddHandler ApplicationData.Current.DataChanged, AddressOf Current_DataChanged
      End Sub

      Private Sub Current_DataChanged(sender As ApplicationData, args As Object)
```

Achtung: Der Aufruf erfolgt aus einem anderen Kontext heraus, Sie müssen zum aktuellen Thread wechseln:

```
         Await Me.Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal,
                 Function()
                    ListBox1.Items.Add("Daten wurden geändert!!!!")
                    Dim roaming As ApplicationDataContainer =
                       ApplicationData.Current.RoamingSettings
                    If roaming.Values("einstellung1") IsNot Nothing Then
                      ListBox1.Items.Add(DirectCast(roaming.Values("einstellung1"), String))
                    End If
                 End Function)
      End Sub
...
```

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.

21.3.9 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 22.1 ab Seite 1081.

21.3.10 Sensible Informationen speichern

An der einen oder anderen Stelle haben wir Sie ja schon darauf aufmerksam gemacht: die Anwendungseinstellungen sind zwar für andere 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 21.17: Credentials speichern, laden und löschen

VB

```
...
Imports 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:

```
        New PasswordVault().Add(New PasswordCredential("MyServer1",
                                                        "MayerMüller", "geheim"))
...
```

Laden der Daten:

```
...
        Dim vault As New PasswordVault()
        Try
```

BEISPIEL 21.17: Credentials speichern, laden und löschen

Wir rufen zunächst die Ressource über ihren Namen ab:

```
            Dim creds = vault.FindAllByResource("MyServer1").FirstOrDefault()
```

Ist diese vorhanden

```
            If creds IsNot Nothing Then
```

bestimmen wir den Nutzernamen:

```
               Dim username As String = creds.UserName
```

Das Abrufen des Passworts gestaltet sich etwas aufwändiger:

```
               Dim password As String = vault.Retrieve("MyServer1", username).Password
               ListBox1.Items.Add(Convert.ToString("Username:") & username)
               ListBox1.Items.Add(Convert.ToString("Passwort:") & password)
            End If
         Catch generatedExceptionName As Exception
            ListBox1.Items.Add("Keine Daten vorhanden!")
         End Try
      End Sub
...
```

Löschen der Daten:

```
...
         Dim 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.

21.4 Praxisbeispiele

21.4.1 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:

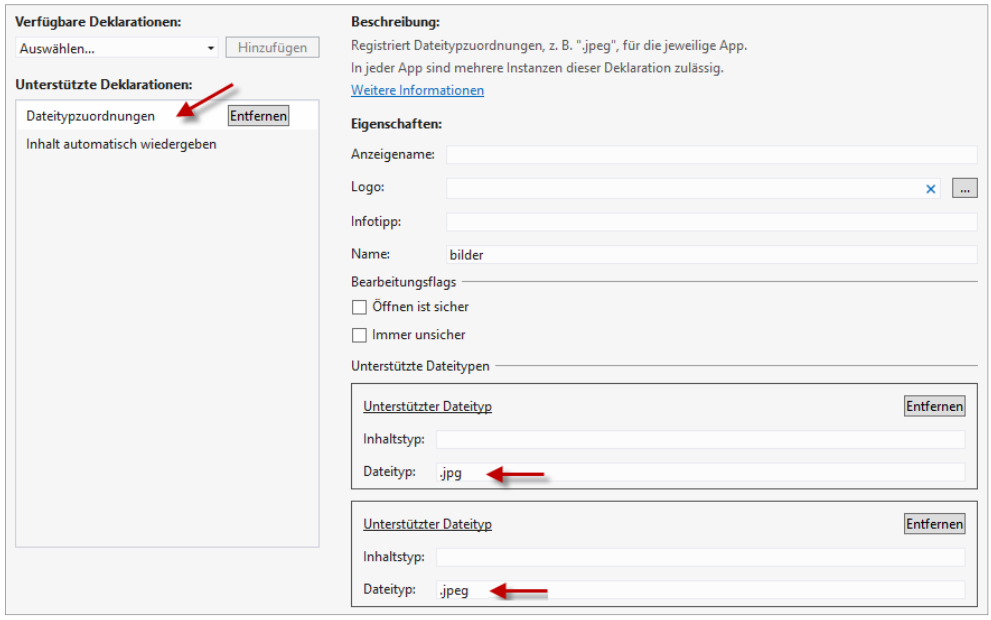
```
\\DCIM\<Liste von Unterordnern>\<Liste der Bilder>
```

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.

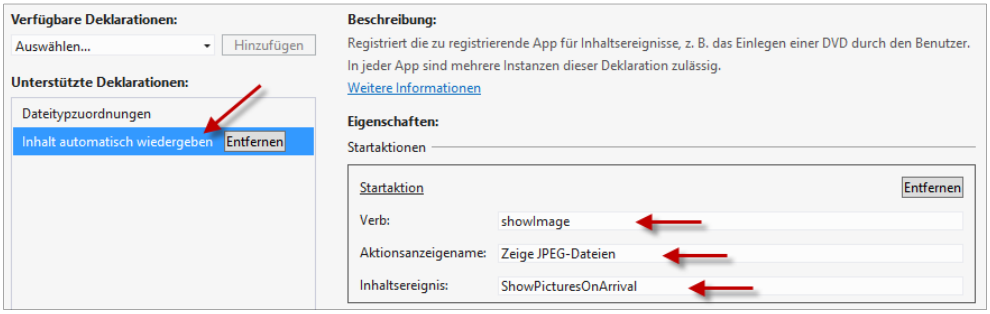
In der Rubrik *Deklarationen* fügen Sie zunächst eine *Datentypzuordnung* hinzu und ergänzen diese, so wie es in der Abbildung zu sehen ist:



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 Auto-play-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:

```
Protected Overrides Async Sub OnFileActivated(args As FileActivatedEventArgs)
```

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]:

```
Dim rootFrame As Frame = Nothing
If Window.Current.Content Is Nothing Then
    rootFrame = New Frame()
    rootFrame.Navigate(GetType(BasicPage1))
    Window.Current.Content = rootFrame
Else
    rootFrame = TryCast(Window.Current.Content, Frame)
End If
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):

```
Dim page1 As BasicPage1 = TryCast(DirectCast(rootFrame.Content, BasicPage1), BasicPage1)
```

[1] Starten Sie die App per Windows 8-Desktop, wird jedoch *OnLaunched* ausgeführt.

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" Then
            page1.ListBox1.Items.Add("OnFileActiveted (args.verb = showImage)")
```

Wir rufen alle übergebenen Verzeichnisse ab (es ist in diesem Fall nur eins):

```
            For Each file In args.Files
                page1.ListBox1.Items.Add("Laufwerk: " + file.Name)
```

Wir ermitteln den Ordner *DCIM* und alle untergeordneten Verzeichnisse:

```
                Dim verz As StorageFolder = TryCast(file, StorageFolder)
                Dim verz_dcim = Await verz.GetFolderAsync("DCIM")
                For Each subverz As StorageFolder In Await verz_dcim.GetFoldersAsync()
```

Alle enthaltenen Dateien (.JPG und .JPEG) auflisten:

```
                    For Each f As StorageFile In Await subverz.GetFilesAsync()
                        page1.ListBox1.Items.Add(f.Path)
                    Next

                Next
            Next
        End If
    End Sub
```

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

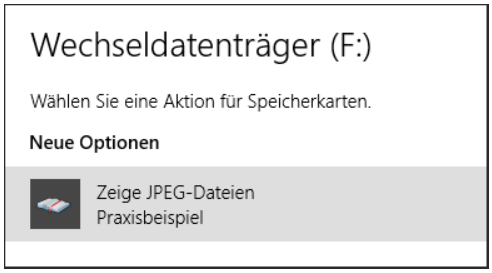
Lediglich eine Seite mit einer enthaltenen *ListBox*. Der Zugriff auf diese muss per *FieldModifier* freigegeben werden, sonst können wir aus der Klasse *App* nicht darauf zugreifen.

```
...
        <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 eine Meldung erscheinen (siehe folgende Abbildung).

HINWEIS: 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.

21.4.2 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 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.vb)

Wir navigieren zum Ersatz-Splash Screen und übergeben das *SplashScreen*-Objekt (das ist der aktuell sichtbare Original-Splash Screen) als Parameter:

```
...
    Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
...
        If rootFrame.Content Is Nothing Then
            If Not rootFrame.Navigate(GetType(ErsatzSplashScreen), args.SplashScreen) Then
                Throw New Exception("Failed to create initial page")
            End If
        End If
        Window.Current.Activate()
    End Sub
```

Quellcode (ErsatzSplashScreen.xaml.vb)

Fügen Sie zunächst folgenden Namespace hinzu:

```
...
Imports Windows.ApplicationModel.Activation
...
Public NotInheritable Class ErsatzSplashScreen
    Inherits 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 Overrides Sub OnNavigatedTo(e As Navigation.NavigationEventArgs)
        Dim splash As SplashScreen = DirectCast(e.Parameter, SplashScreen)
        If splash IsNot Nothing Then
            AddHandler splash.Dismissed, AddressOf splash_Dismissed
        End If
    End Sub
```

Tritt das Ereignis ein, könne wir die Fortschrittsanzeige in Gang setzen (eventuell laden Sie bereits vorher asynchron Daten, die Sie benötigen):

```
    Private Async Sub splash_Dismissed(sender As SplashScreen, args As Object)
        Await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, Sub()
                                                                                    DatenLaden()
                                                                                End Sub)
    End Sub
```

Die Anzeige des Fortschritts:

```
    Public Async Sub DatenLaden()
        For i As Integer = 0 To 10
            Await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Low, Sub()
                                                                                      Progressbar1.Value = i
                                                                                  End Sub)
```

Hier simulieren wir Aufgaben, die von der App zu erledigen sind:

```
            Task.Delay(1000).Wait()
        Next
```

Sind alle Daten geladen, kann das eigentliche Hauptformular gestartet werden:

```
        Me.Frame.Navigate(GetType(BasicPage1))
    End Sub
...
```

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.

21.4.3 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:

```
...
<Applications>
    <Application Id="App"
      Executable="$targetnametoken$.exe"
      EntryPoint="WinApp.App">
      <uap:VisualElements
        DisplayName="WinApp"
        Square150x150Logo="Assets\Logo.png"
        Square44x44Logo="Assets\SmallLogo.png"
        Description="WinApp"
        BackgroundColor="#464646">
        <uap:SplashScreen Image="Assets\SplashScreen.png" />
      </uap:VisualElements>
      <Extensions>
        <uap:Extension Category="windows.fileTypeAssociation">
          <uap:FileTypeAssociation Name="bookreader">
            <uap:DisplayName>Beispielreader</uap:DisplayName>
              <uap:Logo>Assets\Logo.png</uap:Logo>
            <uap:InfoTip>Beispiel für eine Dateiverknüpfung</uap:InfoTip>
            <uap:SupportedFileTypes>
              <uap:FileType>.book</uap:FileType>
            </uap:SupportedFileTypes>
          </uap:FileTypeAssociation>
        </uap:Extension>
      </Extensions>
    </Application>
  </Applications>
...
  <Capabilities>
    <Capability Name="internetClient" />
    <uap:Capability Name="documentsLibrary" />
  </Capabilities>
```

HINWEIS: Sie können Ihre App auch für mehrere Dateitypen registrieren.

Quelltext (App.xaml.vb)

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:

```
...
    Protected Overrides Sub OnFileActivated(args As FileActivatedEventArgs)
```

Der Ablauf entspricht dem bei einem normalen Programmstart:

```
        If args.PreviousExecutionState = ApplicationExecutionState.Running Then
            Window.Current.Activate()
            Return
        End If
        Dim rootFrame = New Frame()
```

Wir übergeben jedoch zusätzlich die Aufrufparameter an die anzuzeigende Seite:

```
        rootFrame.Navigate(GetType(BasicPage1), args)
        Window.Current.Content = rootFrame
        Window.Current.Activate()
        MyBase.OnFileActivated(args)
    End Sub
```

Quelltext (BasicPage.xaml.vb)

Wir die Seite geöffnet, werten wir die Parameter (d h. die Dateiliste) aus:

```
Imports Windows.UI.Xaml.Navigation
Imports Windows.Storage
...
    Protected Overrides Async Sub OnNavigatedTo(e As NavigationEventArgs)
```

Nur wenn die Seite auch mit Parametern gestartet wird können wir etwas auslesen:

```
        If e.Parameter IsNot Nothing Then
```

File-Liste ermitteln:

```
            Dim args As FileActivatedEventArgs = DirectCast(e.Parameter, FileActivatedEventArgs)
```

Jeden Eintrag auswerten und den Dateiinhalt anzeigen:

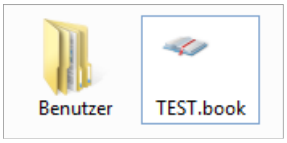
```
            For Each file In args.Files
                ListBox1.Items.Add("Bearbeiten: " + file.Path)
                Dim inhalt As String = Await FileIO.ReadTextAsync(DirectCast(file, StorageFile))
                ListBox1.Items.Add(inhalt)
            Next
        End If
    End Sub
```

HINWEIS: Achten Sie auf die asynchrone Ausführung!

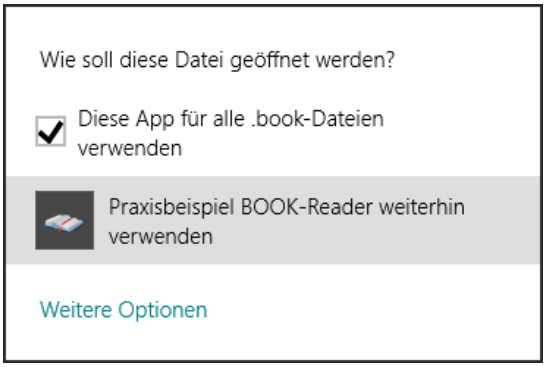
Test

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 (siehe Abbildung).

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 App können Sie die vorliegende App mit folgendem Aufruf starten:

```
Private Async Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
  Application.Current.Exit()
  Dim file As StorageFile = Await KnownFolders.DocumentsLibrary.GetFileAsync("Buchtest.book")
     Await Launcher.LaunchFileAsync(file)
  End Sub
```

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 22

App-Techniken

Dieses Kapitel versteht sich nicht als eine Wiederholung der entsprechenden Microsoft-Dokumentation, sondern als Hilfe für die Lösung spezifischer Programmieraufgaben im Umfeld einer Windows 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.

22.1 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 21.3 ab Seite1058 besprochen.

22.1.1 Verzeichnisinformationen auflisten

Haben Sie es geschafft ein *StorageFolder*-Objekt zu initialisieren (Abschnitt 21.3), können Sie dieses nutzen, um mehr über den Ordner zu erfahren.

BEISPIEL 22.1: Abrufen von Informationen über ein *StorageFolder*-Objekt

VB
```
Imports Windows.Storage
...
    Private Async Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
```

BEISPIEL 22.1: 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:

```
Dim folder As StorageFolder = 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:

```
Dim prop = Await folder.GetBasicPropertiesAsync()
ListBox1.Items.Add("Properties.DateModified: " + prop.DateModified.ToString())
ListBox1.Items.Add("Properties.ItemDate: " + prop.ItemDate.ToString())
ListBox1.Items.Add("Properties.Size: " + prop.Size.ToString())
```

Nicht genug der Pein, es wartet noch eine Eigenschaftenauflistung auf Sie:

```
Dim props = Await prop.RetrievePropertiesAsync(New List(Of String)() From { _
})
For Each p In props
    If p.Value IsNot Nothing Then
        ListBox1.Items.Add("RetrievePropertiesAsync: " & p.Key & ": " & _
                    p.Value.ToString())
    End If
Next
End Sub
...
```

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: 14.08.2015 11:12:12 +02:00
Properties.ItemDate: 14.08.2015 11:12:12 +02:00
Properties.Size: 0
RetrievePropertiesAsync: System.KindText: Ordner
RetrievePropertiesAsync: System.ZoneIdentifier: 3
```

BEISPIEL 22.1: Abrufen von Informationen über ein *StorageFolder*-Objekt

```
RetrievePropertiesAsync: System.Shell.SFGAOFlagsStrings: System.String()
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*

22.1.2 Unterverzeichnisse auflisten

Nichts leichter als das, rufen Sie die enthaltenen Ordner per *GetFoldersAsync*-Methode ab:

BEISPIEL 22.2: Unterverzeichnisse auflisten

VB
```
...
Imports Windows.Storage
...
    Private Async Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        Dim folders As IReadOnlyList(Of StorageFolder) = _
                Await ApplicationData.Current.LocalFolder.GetFoldersAsync()
        For Each folder In folders
            ListBox1.Items.Add(folder.Path)
        Next
   End Sub
...
```

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 22.3: Rekursive Suche nach Verzeichnissen

VB
```
...
Imports Windows.Storage
Imports Windows.Storage.Search
...
    Private Async Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
```

Eine *QueryOptions*-Instanz erzeugen:

```
        Dim queryOptions As 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 22.3: Rekursive Suche nach Verzeichnissen

Die Abfrage erstellen (bis jetzt passiert im Dateisystem noch gar nichts):

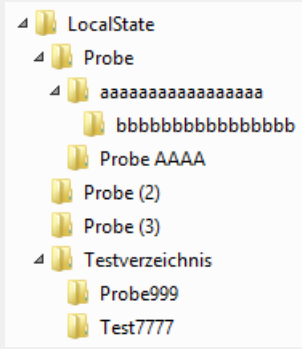
```
        Dim query = _
            ApplicationData.Current.LocalFolder.CreateFolderQueryWithOptions(queryOptions)
```

Jetzt das Verzeichnis abrufen:

```
        For Each f In Await query.GetFoldersAsync()
            ListBox1.Items.Add(f.Path)
        Next
    End Sub
...
```

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

► 22.7.1 Ein Verzeichnis auf Änderungen überwachen

22.1.3 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 22.4: Einige Unterverzeichnisse im App-Datenverzeichnis erstellen

VB

```
...
        Await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                    CreationCollisionOption.ReplaceExisting)
        Await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                    CreationCollisionOption.ReplaceExisting)
...
```

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 22.5: Verzeichnis erstellen und löschen

VB

```
...
    Private Async Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
```

Erzeugen:

```
        Await ApplicationData.Current.LocalFolder.CreateFolderAsync("Probe",
                        CreationCollisionOption.ReplaceExisting)
```

Auswahl und Löschen:

```
        Dim folder As StorageFolder =
              Await ApplicationData.Current.LocalFolder.GetFolderAsync("Probe")
```

[1] Diese Variante funktionierte bei den Autoren nicht fehlerfrei.

BEISPIEL 22.5: Verzeichnis erstellen und löschen

```
        Await folder.DeleteAsync(StorageDeleteOption.PermanentDelete)
    End Sub
...
```

HINWEIS: Über den Parameter *StorageDeleteOption* bestimmen Sie, ob das Verzeichnis endgültig (*PermanentDelete*) oder per Papierkorb (*Default*) gelöscht wird.

22.1.4 Dateien auflisten

Die grundsätzliche Vorgehensweise entspricht der im Abschnitt 22.1.2, das folgende Beispiel zeigt drei Varianten:

BEISPIEL 22.6: Dateien auflisten

VB

```
...
Imports Windows.Storage
Imports Windows.Storage.Search
...
```

Die einfachste Variante, alle Dateien eines Ordners zu ermitteln:

```
Dim files As IReadOnlyList(Of StorageFile) = _
            Await ApplicationData.Current.LocalFolder.GetFilesAsync()
Debug.WriteLine(ApplicationData.Current.LocalFolder.Path)
For Each file In files
    ListBox1.Items.Add(file.Path)
Next
```

Um nach Dateien rekursiv (mit Unterverzeichnissen) suchen zu können, müssen Sie ein *QueryOptions*-Objekt erstellen und parametrieren:

```
Dim queryOptions As New QueryOptions(CommonFileQuery.DefaultQuery, Nothing)
queryOptions.FolderDepth = FolderDepth.Deep
queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable
```

Mit den *QueryOptions* wird eine Query erstellt, die dann mit *GetFilesAsync* abgefragt wird:

```
Dim query = _
     ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions)
For Each f In Await query.GetFilesAsync()
    ListBox1.Items.Add(f.Path)
Next
```

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.OrderByName, New String() {".txt"})
```

BEISPIEL 22.6: Dateien auflisten

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
        Dim query = _
            ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions)
```

Anzeige:

```
        For Each f In Await query.GetFilesAsync()
            ListBox1.Items.Add(f.Path)
        Next
...
```

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:

```
Dim queryOptions As 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:

```
Dim query = ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions)
```

6. Dateiliste abrufen:

```
Dim liste = Await query.GetFilesAsync()
```

7. Liste filtern (14 Tage) und sortieren nach Erstellungsdatum:

```
For Each f In liste.Where(Function(ff)
    (System.DateTime.Now - ff.DateCreated).Days <= 14).OrderBy(Function(ff) ff.DateCreated)
    ListBox1.Items.Add(f.Path)
Next
```

HINWEIS: Die Funktion sucht natürlich nur im App-Datenverzeichnis, eine komplette Laufwerkssuche ist für 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*).
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.

22.1.5 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 21.3.7, Seite 1064) 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 22.7: Eine neue leere Datei erstellen

VB
```
...
Imports Windows.Storage
...
    Private Async Sub Button_Click_7(sender As Object, e As RoutedEventArgs)
        Dim folder As StorageFolder = ApplicationData.Current.LocalFolder
        Dim file As StorageFile = Await folder.CreateFileAsync("daten.txt",
                                        CreationCollisionOption.OpenIfExists)

   End Sub
...
```

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 22.8: Einen Text an eine Textdatei anhängen

VB
```
...
Imports Windows.Storage
...
        Dim folder As StorageFolder = ApplicationData.Current.LocalFolder
```

BEISPIEL 22.8: Einen Text an eine Textdatei anhängen

```
        Dim file As StorageFile = 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.

```
        Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... " & vbCr & vbLf)
        Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... " & vbCr & vbLf)
```

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 22.9: Schreiben einer Textdatei per *StreamWriter*

VB

```
...
Imports Windows.Storage
...
        Dim folder As StorageFolder = ApplicationData.Current.LocalFolder
        Dim file As StorageFile = Await folder.CreateFileAsync("daten.txt",
                                      CreationCollisionOption.OpenIfExists)
        Using writer As 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)")
```

BEISPIEL 22.9: Schreiben einer Textdatei per *StreamWriter*

```
            Await writer.WriteLineAsync("Noch mehr Zeilen (6)")
        End Using
    End Sub
...
```

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 22.10: Textdatei aus dem Installationsverzeichnis komplett einlesen

VB

```
...
Imports Windows.Storage
...
    Private Async Sub Button_Click_8(sender As Object, e As RoutedEventArgs)
```

Installationspfad bestimmen:

```
        Dim folder As StorageFolder = _
                  Windows.ApplicationModel.Package.Current.InstalledLocation
        Dim file As StorageFile = Await folder.GetFileAsync("TextFile.txt")
```

Die komplette Datei in eine *IList(Of String)* einlesen:

```
        Dim inhalt = Await FileIO.ReadLinesAsync(file)
```

Zeilen anzeigen:

```
        For Each item In inhalt
            ListBox1.Items.Add(item)
        Next
    End Sub
...
```

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 22.11: Textdatei zeilenweise per *StreamReader* einlesen

VB

```
...
        Dim folder As StorageFolder = _
                      Windows.ApplicationModel.Package.Current.InstalledLocation
```

BEISPIEL 22.11: Textdatei zeilenweise per *StreamReader* einlesen

```
        Dim file As StorageFile = Await folder.GetFileAsync("TextFile.txt")
        Dim zeile As String = ""
```

StreamReader erzeugen:

```
        Using reader As New StreamReader(Await file.OpenStreamForReadAsync())
```

Zeilenweise einlesen:

```
            While (InlineAssignHelper(zeile, Await reader.ReadLineAsync())) IsNot Nothing
                ListBox1.Items.Add(zeile)
            End While
        End Using
...
```

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.

22.1.6 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 22.12: Datei kopieren

VB

```
...
    Private Async Sub Button_Click_9(sender As Object, e As RoutedEventArgs)
```

Quell-Verzeichnis bestimmen:

```
        Dim folder As StorageFolder = ApplicationData.Current.LocalFolder
```

Quelldatei bestimmen:

```
        Dim file As StorageFile = Await folder.CreateFileAsync("demodatei.dat",
                 CreationCollisionOption.ReplaceExisting)
```

In diesem Fall Zielverzeichnis erzeugen:

```
        Dim zielfolder As StorageFolder = Await folder.CreateFolderAsync("Backup",
                         CreationCollisionOption.OpenIfExists)
```

Datei kopieren:

```
        Await file.CopyAsync(zielfolder, DateTime.Now.ToString("d") + "_demodatei.dat",
                             NameCollisionOption.ReplaceExisting)
    End Sub
```

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 22.13: Datei verschieben

VB

```
...
    Private Async Sub Button_Click_10(sender As Object, e As RoutedEventArgs)
```

Quellverzeichnis ermitteln:

```
        Dim folder As StorageFolder = ApplicationData.Current.LocalFolder
```

Datei öffnen:

```
        Dim file As StorageFile = Await folder.CreateFileAsync("demodatei.dat",
                                      CreationCollisionOption.ReplaceExisting)
```

Zielverzeichnis erzeugen oder öffnen:

```
        Dim zielfolder As StorageFolder = Await folder.CreateFolderAsync("Backup",
                                        CreationCollisionOption.OpenIfExists)
```

Datei verschieben:

```
        Await file.MoveAsync(zielfolder, DateTime.Now.ToString("d") + "_demodatei.dat",
                             NameCollisionOption.ReplaceExisting)
    End Sub
...
```

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.

22.1.7 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*

Wer von Windows 8/8.1 kommt, wird jetzt intuitiv an die riesigen Dateidialog denken, die zwar für die Toucheingabe sinnvoll, für den normalen Desktop-Nutzer aber völlig fehl am Platz waren. Doch da liegen Sie falsch, auf dem Windows-Desktop werden Ihnen jetzt die gewohnten Dialoge angezeigt, wie Sie diese von normalen Desktop-Anwendungen her kennen.

HINWEIS: Ein Test auf Windows Phone war den Autoren zum Zeitpunkt der Bucherstellung leider nicht möglich, es steht jedoch zu vermuten, dass in diesem Fall ein gerätespezifischer Dateidialog angezeigt wird.

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 22.14: Dateiauswahl mit *FileOpenPicker*

VB

```
...
Imports Windows.Storage.Pickers
...
    Private Async Sub Button_Click_11(sender As Object, e As RoutedEventArgs)
```

Picker instanziieren:

```
        Dim fileopen As New FileOpenPicker()
```

BEISPIEL 22.14: Dateiauswahl mit *FileOpenPicker*

Filter festlegen:

```
        fileopen.FileTypeFilter.Add(".png")
        fileopen.FileTypeFilter.Add(".bmp")
```

Dialog öffnen:

```
        Dim file As StorageFile = Await fileopen.PickSingleFileAsync()
```

Auswerten:

```
        If file IsNot Nothing Then
            ListBox1.Items.Add("Auswahl: " + file.Path)
            Await file.CopyAsync(ApplicationData.Current.LocalFolder)
        End If
    End Sub
...
```

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 22.15: Mehrfachauswahl mit *FileOpenPicker*

VB

```
...
        Dim fileopen As 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:

```
        Dim files = Await fileopen.PickMultipleFilesAsync()
        If files IsNot Nothing Then
            For Each file1 In files
                ListBox1.Items.Add("Auswahl: " + file1.Path)
            Next
        End If
...
```

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 1099).

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 22.16: Datei mit *FileSavePicker* speichern

VB

```
...
Imports Windows.Storage.Pickers
...
    Private Async Sub Button_Click_12(sender As Object, e As RoutedEventArgs)
```

Instanz erzeugen:

```
        Dim filesave As New FileSavePicker()
```

Dateitypauswahl festlegen (diese filtern auch die Ansicht):

```
        filesave.FileTypeChoices.Add("Vektorgrafik", New List(Of String)() From { _
            ".wmf",
```

BEISPIEL 22.16: Datei mit *FileSavePicker* speichern

```
            ".emf" _
        })
        filesave.FileTypeChoices.Add("Pixelgrafik", New List(Of String)() From { _
            ".jpg",
            ".png" _
        })
```

Die Standard-Extension festlegen:

```
        filesave.DefaultFileExtension = ".wmf"
```

Einen Filenamen vorgeben (dieser kann editiert werden):

```
        filesave.SuggestedFileName = "Karte.wmf"
```

Dialog anzeigen:

```
        Dim file As StorageFile = Await filesave.PickSaveFileAsync()
```

Rückgabewert auswerten:

```
        If file IsNot Nothing Then
            ListBox1.Items.Add("Auswahl: " + file.Path)
            Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben ... " & vbCr & vbLf)
        End If
    End Sub
...
```

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 22.17: Einen *FolderPicker* zur Verzeichnisauswahl einsetzen

VB

```
...
    Private Async Sub Button_Click_13(sender As Object, e As RoutedEventArgs)
```

Instanz erzeugen:

```
        Dim folderpic As New FolderPicker()
```

Alle Dateien anzeigen:

```
        folderpic.FileTypeFilter.Add("*")
```

BEISPIEL 22.17: Einen *FolderPicker* zur Verzeichnisauswahl einsetzen

Listenansicht auswählen:

```
folderpic.ViewMode = PickerViewMode.List
```

Wir starten in der Bild-Bibliothek:

```
folderpic.SuggestedStartLocation = PickerLocationId.PicturesLibrary
```

Dialog anzeigen:

```
Dim folder As StorageFolder = Await folderpic.PickSingleFolderAsync()
```

Dialog auswerten:

```
If folder IsNot Nothing Then
```

Eine Datei im ausgewählten Verzeichnis erstellen:

```
        Dim file As StorageFile = Await folder.CreateFileAsync("daten.txt",
                                  CreationCollisionOption.OpenIfExists)
```

Wir füllen die Datei mit sinnlosem Inhalt:

```
        Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben" & vbCr & vbLf)
        Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben" & vbCr & vbLf)
        Await FileIO.AppendTextAsync(file, "Mal etwas Text schreiben" & vbCr & vbLf)
```

Alle Dateien dieses Verzeichnisses anzeigen:

```
        For Each f In Await folder.GetFilesAsync()
            ListBox1.Items.Add(f.Path)
```

Auch das geht jetzt – eine Datei löschen:

```
            If f.Name = "EBooks.txt" Then Await f.DeleteAsync()
        Next
    End If
End Sub
...
```

22.1.8 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 22.18: Verwendung der *FutureAccessList*

VB

```
...
Imports Windows.Storage.AccessCache
...
```

BEISPIEL 22.18: Verwendung der *FutureAccessList*

Stellen Sie sich vor, Sie rufen mit *PickSingleFolderAsync* einen Zielordner auf (z.B. den Desktop):

```
        Dim folder As StorageFolder = Await folderpic.PickSingleFolderAsync()
        If folder IsNot Nothing Then
...
```

An dieser Stelle speichern Sie den *StorageFolder* in der *FutureAccessList* ab:

```
            Dim token As String = 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:

```
    Private Async Sub Button_Click_14(sender As Object, e As RoutedEventArgs)
        ListBox1.Items.Add("Vorhandene FutureAccessList-Entries")
        For Each 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:

```
            Dim folder As StorageFolder = _
            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:

```
            Dim file As StorageFile = Await folder.CreateFileAsync("daten.txt",
                                        CreationCollisionOption.OpenIfExists)
            Await FileIO.WriteTextAsync(file, "Ich wurde gerade verändert!")
        Next
    End Sub
```

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>) Then
   Dim folder As StorageFolder = _
       Await  StorageApplicationPermissions.FutureAccessList.GetFolderAsync(<GespeicherteID>)
End If
```

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].

22.1.9 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 22.19: Eine Liste von Dateien zur MRU hinzufügen

VB
```
...
Imports Windows.Storage
Imports Windows.Storage.Pickers
Imports Windows.Storage.AccessCache
...
    Private Async Sub Button_Click_15(sender As Object, e As RoutedEventArgs)
        Dim fileopen As New FileOpenPicker()
        fileopen.FileTypeFilter.Add("*")
        fileopen.ViewMode = PickerViewMode.Thumbnail
        fileopen.SuggestedStartLocation = PickerLocationId.PicturesLibrary
```

[1] Sie können ja den Anwender "überreden" die Root von Laufwerk "C:" auszuwählen ...

BEISPIEL 22.19: Eine Liste von Dateien zur MRU hinzufügen

```
        fileopen.CommitButtonText = "Zur MRU hinzufügen"
```

Wir wählen per Picker mehrere Dateien aus:

```
        Dim files = Await fileopen.PickMultipleFilesAsync()
        If files IsNot Nothing Then
```

Einfügen in die MRU-Liste (wir fügen einen Zeitstempel mit ein, so sehen Sie gleich, wie die Einträge abgelegt werden):

```
            For Each file In files
                StorageApplicationPermissions.MostRecentlyUsedList.Add(file,
                        DateTime.Now.ToString() + "  " + file.Path)
            Next
        End If
    End Sub
```

Das spätere Auslesen:

```
    Private Sub Button_Click_16(sender As Object, e As RoutedEventArgs)
        ListBox1.Items.Clear()
        For Each entry As AccessListEntry In _
                StorageApplicationPermissions.MostRecentlyUsedList.Entries
            ListBox1.Items.Add(entry.Metadata)
        Next
    End Sub
...
```

Damit schließen wir die Thematik "Dateien und Verzeichnisse" ab und wenden uns einigen neuen Konzepten der WinRT-Programmierung zu.

22.2 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.

22.2.1 Zwischenablage

Auch wenn es für 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 Zwischenab-

lage kopieren. Die Zielanwendung entscheidet dann (bzw. der Nutzer) welches der unterstützten Formate eingefügt werden soll.

BEISPIEL 22.20: Kopieren und Einfügen von Texten

VB

```
Imports Windows.ApplicationModel.DataTransfer
...
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
```

Zunächst unser Text:

```
        Dim meintext As String = "Dieser Text stammt aus der App!!!!"
```

Gleiches nochmal im HTML-Format:

```
        Dim meinhtmltext As String = ("<h1>" & meintext) + "</h1><br/> Noch eine Zeile!"
```

Wir erstellen ein *DataPackage*, dieses nimmt die Daten auf:

```
        Dim datapackage As New DataPackage()
```

Die Texte zum *DataPackage* hinzufügen:

```
        datapackage.SetHtmlFormat(HtmlFormatHelper.CreateHtmlFormat(meinhtmltext))
        datapackage.SetText(meintext)
```

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 ex As Exception
            ListBox1.Items.Add("Fehler: " + ex.Message)
        End Try
    End Sub
...
```

Beim Abrufen von Daten lesen wir zunächst ein *DataPackageView*-Objekt ein:

```
    Private Async Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        Dim dpv As DataPackageView = Clipboard.GetContent()
```

Nachfolgend prüfen wir, ob das gewünschte Datenformat (in unserem Fall Text) vorhanden ist:

```
        If dpv.Contains(StandardDataFormats.Text) Then
```

Ist dies der Fall, können wir die Daten per *GetTextAsync* abrufen:

```
            Dim txt As String = Await dpv.GetTextAsync()
            ListBox1.Items.Add(txt)
        End If
    End Sub
```

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 22.21: Kopieren und Einfügen von Bildern

VB

```
...
Imports Windows.ApplicationModel.DataTransfer
Imports Windows.Storage.Streams
Imports Windows.UI.Xaml.Media.Imaging
...
    Private Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
```

DataPackage erzeugen:

```
        Dim datapackage As 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)
    End Sub
...
```

Beim Einfügen der Daten, in diesem Fall in ein *Image*-Control zur Anzeige, müssen wir ebenfalls den Umweg über eine *IRandomAccessStreamReference* nehmen:

```
   Private Async Sub Button_Click_5(sender As Object, e As RoutedEventArgs)
        Dim dpv As DataPackageView = Clipboard.GetContent()
```

Wenn eine Bitmap enthalten ist:

```
        If dpv.Contains(StandardDataFormats.Bitmap) Then
            Try
```

Bitmap-Inhalte abrufen:

```
                Dim imgdata As IRandomAccessStreamReference = Await dpv.GetBitmapAsync()
                If imgdata IsNot Nothing Then
```

Ein internes *BitmapImage*-Objekt wird mit den Daten initialisiert und nachfolgend als Quelle für das *Image*-Control genutzt:

```
                    Dim img As New BitmapImage()
                    img.SetSource(Await imgdata.OpenReadAsync())
```

BEISPIEL 22.21: Kopieren und Einfügen von Bildern

```
                    Image1.Source = img
                End If
            Catch ex As Exception
                ListBox1.Items.Add(ex.Message)
            End Try
        End If
    End Sub
...
```

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 22.22: Dateien in die Zwischenablage kopieren

VB

```
...
Imports Windows.ApplicationModel.DataTransfer
Imports Windows.Storage.Streams
Imports Windows.Storage
Imports Windows.Storage.Pickers
...
    Private Async Sub Button_Click_6(sender As Object, e As RoutedEventArgs)
```

Zunächst wählen wir einige Dateien aus der Bild-Bibliothek aus:

```
        Dim fileopen As New FileOpenPicker()
        fileopen.FileTypeFilter.Add("*")
        fileopen.ViewMode = PickerViewMode.Thumbnail
        fileopen.SuggestedStartLocation = PickerLocationId.PicturesLibrary
        fileopen.CommitButtonText = "Zur Zwischenablage hinzufügen"
        Dim files = Await fileopen.PickMultipleFilesAsync()
        If files.Count > 0 Then
```

Wurden Dateien gewählt, erzeugen wir das bereits bekannte *DataPackage* und übergeben die *StorageFiles*:

```
            Dim 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
```

BEISPIEL 22.22: Dateien in die Zwischenablage kopieren

```
            Clipboard.SetContent(datapackage)
        Catch ex As Exception
            ListBox1.Items.Add(ex.Message)
        End Try
    End If
End Sub
```

Wollen wir selbst Dateien abrufen, prüfen wir zunächst auf das Vorhandensein des richtigen Datenformats:

```
Private Async Sub Button_Click_7(sender As Object, e As RoutedEventArgs)
    Dim dpv = Clipboard.GetContent()
    If dpv.Contains(StandardDataFormats.StorageItems) Then
        Try
```

Wir rufen die Liste der Dateien (oder auch Verzeichnisse) ab:

```
            Dim files = Await dpv.GetStorageItemsAsync()
            If files.Count > 0 Then
```

Wir bestimmen die auszuführende Operation (hier nur zur Anzeige):

```
                Dim todo As String = dpv.RequestedOperation.ToString()
```

Wir listen die Dateien auf (hier würde eigentlich eine Auswertung erfolgen):

```
                For Each file In files
                    If file.IsOfType(StorageItemTypes.Folder) Then
                        ListBox1.Items.Add("Folder: " + file.Path)
                    End If
                    ListBox1.Items.Add((todo & Convert.ToString(": ")) + file.Path)
                Next
            End If
        Catch ex As Exception
            ListBox1.Items.Add(ex.Message)
        End Try
    End If
End Sub
```

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.

```
...
                Dim files = Await dpv.GetStorageItemsAsync()
                If (files.Count > 0)
                For Each file In files
                    If file.IsOfType(StorageItemTypes.Folder) Then
```

```
                        ListBox1.Items.Add("Folder: " + file.Path)
                    End If
                    ListBox1.Items.Add((todo & Convert.ToString(": ")) + 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 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 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 22.23: Reaktion auf Änderungen in der Zwischenablage

VB

```
...
Imports Windows.ApplicationModel.DataTransfer
...
Public NotInheritable Class BasicPage1
    Inherits Common.LayoutAwarePage

    Private dtm As DataTransferManager

    Public Sub New()
        Me.InitializeComponent()
```

Die Ereignisprozedur zuweisen:

```
        AddHandler Clipboard.ContentChanged, AddressOf Clipboard_ContentChanged
    End Sub
```

Die eigentliche Ereignisprozedur:

```
    Private Sub Clipboard_ContentChanged(sender As Object, e As Object)
        ListBox1.Items.Add("Änderung in der Zwischenablage: ")
```

Enthaltene Formate auslesen und anzeigen:

```
        Dim dpv As DataPackageView = Clipboard.GetContent()
        For Each format In dpv.AvailableFormats
            ListBox1.Items.Add("   verfügbar: " + format.ToString())
```

BEISPIEL 22.23: Reaktion auf Änderungen in der Zwischenablage

```
        Next
    End Sub
...
```

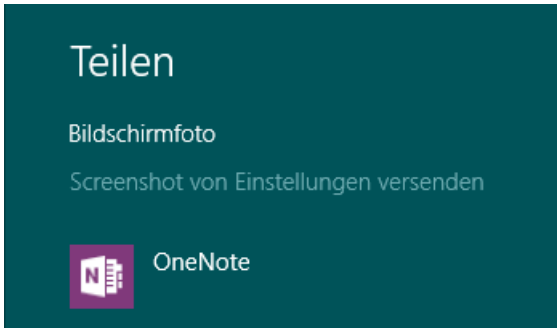
HINWEIS: Hier tut sich dann auch eine kleine Schwachstelle des WinRT-Sicherheitskonzepts auf: Sie überwachen beispielsweise die Zwischenablage und sammeln vielleicht interessante *StorageFile*- oder *StorageFolder*-Objekte ein und speichern diese in der *FutureAccessList*. Später haben Sie dann vollen Zugriff auf den entsprechenden Speicherort (so er noch vorhanden ist).

Kommen wir nun jedoch zu den modernen Nachfolgern der Zwischenablage.

22.2.2 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 (früher mal CharmBar, jetzt per *Win+H* erreichbar):



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 22.24: Daten mit dem *DataTransferManager* teilen

VB

```
...
Imports Windows.ApplicationModel.DataTransfer
Imports Windows.Storage.Streams
Imports Windows.UI.Xaml.Media.Imaging
Imports Windows.Storage
...
Public NotInheritable Class BasicPage1
    Inherits Common.LayoutAwarePage
```

Eine lokale Instanz des *DataTransferManager* erzeugen:

```
    Private dtm As DataTransferManager
```

Wird die Seite geöffnet, rufen wir für die aktuelle Ansicht den *DataTransferManager* ab und belegen das *DataRequested*-Ereignis:

```
    Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
        dtm = DataTransferManager.GetForCurrentView()
        AddHandler dtm.DataRequested, AddressOf DataRequested
    End Sub
```

Beim Schließen der Seite sollten wir auch das Ereignis wieder "abklemmen":

```
    Protected Overrides Sub OnNavigatedFrom(e As NavigationEventArgs)
        RemoveHandler dtm.DataRequested, AddressOf DataRequested
    End Sub
```

Das folgende Ereignis wird durch die Teilen-Funktion auf dem Charmbar aktiviert:

```
    Private Async Sub DataRequested(sender As DataTransferManager,
                                    args As DataRequestedEventArgs)
```

Um das *DataPackage* müssen wir uns nicht selbst kümmern, dieses wird gleich per Parameter übergeben:

```
        Dim datapackage As 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.")
```

BEISPIEL 22.24: Daten mit dem *DataTransferManager* teilen

Eine Bitmap kopieren wir gleich mit:

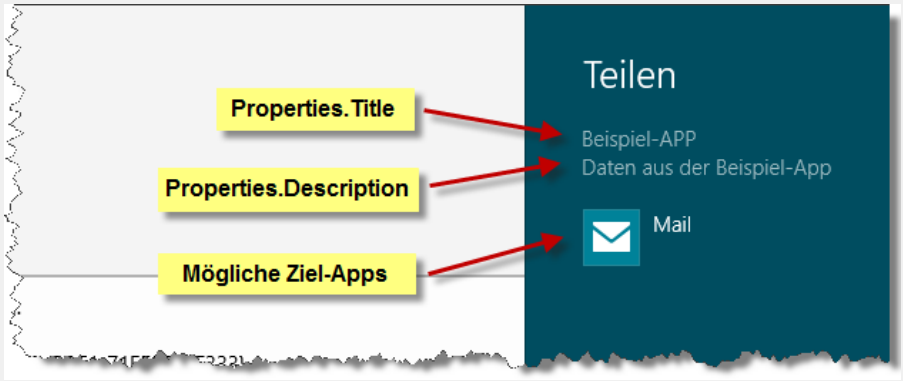
```
        datapackage.SetBitmap(RandomAccessStreamReference.CreateFromUri(
                              New Uri("ms-appx:///bulboff.png"))
```

Alternativ können Sie auch StorageFiles teilen:

```
        Dim items As New List(Of IStorageItem)()
        items.Add(Await Package.Current.InstalledLocation.GetFileAsync("bulbon.png"))
            datapackage.SetStorageItems(items)
        datapackage.SetStorageItems(items)
    End Sub
...
```

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:

[1] Im Augenblick ist diese Liste noch sehr überschaubar, im Laufe der Zeit werden sich sicher diverse Apps auf den PCs tummeln.

BEISPIEL 22.25: Teilen per App aktivieren

VB
```
...
    Private Sub Button_Click_8(sender As Object, e As RoutedEventArgs)
        DataTransferManager.ShowShareUI()
    End Sub
...
```

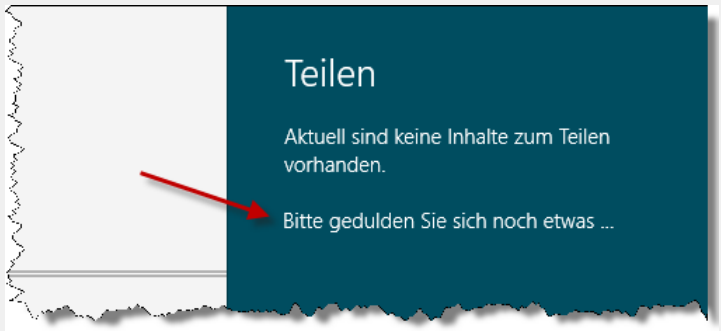
Sie möchten temporär keine Daten teilen

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 22.26: Meldung, wenn keine Daten vorhanden sind

VB
```
...
    Private Async Sub DataRequested(sender As DataTransferManager,
                                    args As DataRequestedEventArgs)
        Dim datapackage As 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 ...")
    End Sub
...
```

Ergebnis



Damit beenden wir unsere Ausführungen zum Teilen von Daten und sehen uns die Gegenseite, d h. den Empfänger der Daten, näher an.

22.2.3 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:

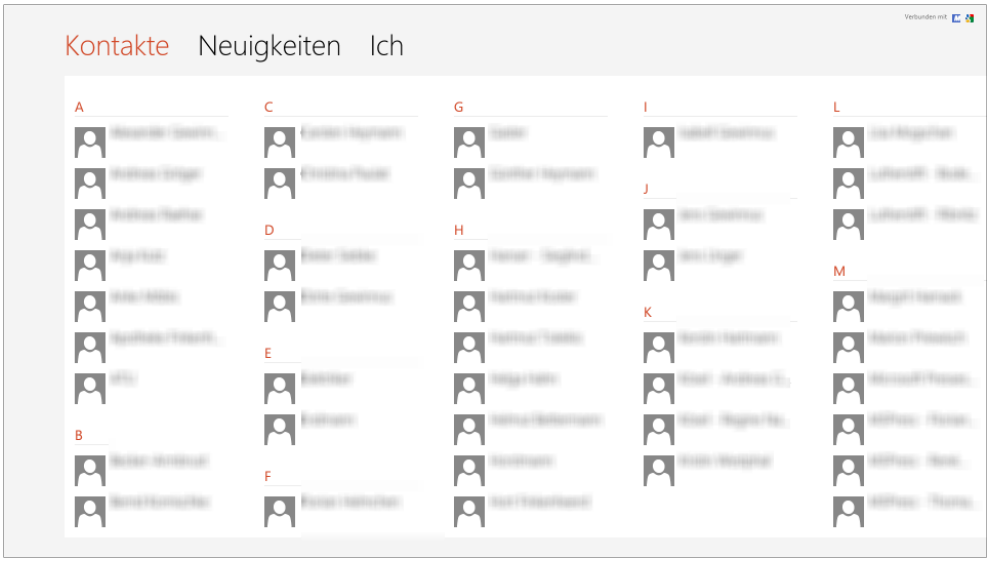
- Registrieren der App als Freigabeziel
- Definieren der unterstützten Daten-/Dateiformate
- Implementieren des Eintrittspunktes *OnShareTargetActivated*
- Definieren einer Seite für die Anzeige und Übernahme der Daten
- Auswerten der Daten

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:

► 22.7.2 Eine App als Freigabeziel verwenden

22.2.4 Zugriff auf die Kontaktliste

Windows 8/10 bietet eine zentrale Kontaktliste, die die Daten verschiedener Konten (Outlook.com, 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 22.27: Verwendung *ContactPicker*

VB

Namespace für den *ContactPicker* einbinden:

```
Imports Windows.ApplicationModel.Contacts
```

BEISPIEL 22.27: Verwendung *ContactPicker*

```
    Private Async Sub Button_Click_9(sender As Object, e As RoutedEventArgs)
```

Picker instanziieren:

```
        Dim picker As New ContactPicker()
```

Wir möchten mehrere Kontakte abrufen, alternativ können Sie auch die Methode *PickSingleContactAsyc* aufrufen:

```
        Dim contacts = Await picker.PickMultipleContactsAsync()
```

Wir werten die Liste der zurückgegebenen Kontakte aus:

```
        For Each 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:

```
            For Each email In contact.Emails
                ListBox1.Items.Add("    Email: " + email.Value)
            Next
            For Each phonenumber In contact.PhoneNumbers
                ListBox1.Items.Add("    Telefon: " + phonenumber.Category + _
                                                 " " + phonenumber.Value)
            Next
```

Das *Value*-Feld enthält die eigentliche Emailadresse bzw. Telefonnummer, die Beschriftung (Privat, Geschäftlich Mobil etc.) findet sich im Feld *Category*.

```
        Next
    End Sub
...
```

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.

22.3 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.

22.3.1 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.

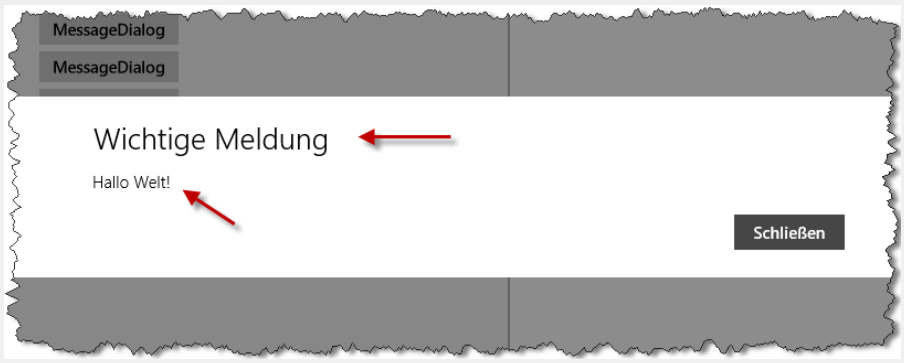
Prinzipiell gibt es die Klasse bzw. deren Funktionalität nicht mehr, Sie müssen sich wohl oder übel mit der neuen Klasse *MessageDialog* anfreunden, die eine ähnliche Funktionalität aufweist.

BEISPIEL 22.28: Einfache Meldung per MessageDialog anzeigen

VB

```
Imports Windows.UI.Popups
...
    Private Async Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Dim msgdlg As New MessageDialog("Hallo Welt!", "Wichtige Meldung")
        Dim res = Await msgdlg.ShowAsync()
    End Sub
...
```

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 22.29: *MessageDialog* mit Auswahl

VB

...

Dialog erzeugen:

```
Dim msgdlg As 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", nothing, "t1"))
msgdlg.Commands.Add(New UICommand("Taste 2", nothing, "t2"))
```

Taste 1 mit der Enter-Taste verbinden:

```
msgdlg.DefaultCommandIndex = 0
```

Taste 2 mit der Escape-Taste verbinden:

```
msgdlg.CancelCommandIndex = 1
```

Dialoganzeige:

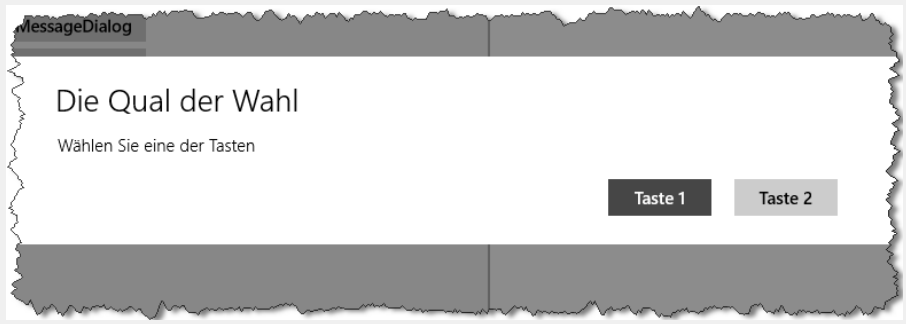
```
Dim res = Await msgdlg.ShowAsync()
```

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 22.30: Auswahl per Lambda-Ausdruck auswerten

VB

```
...
    Private Async Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        Dim msgdlg As 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", Sub(c)
                      ListBox1.Items.Insert(0, "Taste [" + c.Label + "] wurde gedrückt!")
                    End Sub))
        msgdlg.Commands.Add(New UICommand("Taste 2", Sub(c)
                      ListBox1.Items.Insert(0, "Taste [" + c.Label + "] wurde gedrückt!")
                    End Sub))
        msgdlg.DefaultCommandIndex = 0
        msgdlg.CancelCommandIndex = 1
        Dim res = Await msgdlg.ShowAsync()
        ListBox1.Items.Insert(0, "Gewählt: " + res.Id)
    End Sub
...
```

HINWEIS: Beachten Sie, dass bei obigem Beispiel keine CommandId zurückgegeben wird, wir haben auch keine definiert.

BEISPIEL 22.31: Auswahl per Methode auswerten

VB

```
...
    Private Async Sub Button_Click_5(sender As Object, e As RoutedEventArgs)
        Dim msgdlg As 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(AddressOf Me.CommandInvokedHandler), "MeineID"))
```

Hier ohne CommandId:

```
        msgdlg.Commands.Add(New UICommand("Taste 2",
            New UICommandInvokedHandler(AddressOf Me.CommandInvokedHandler)))
        msgdlg.DefaultCommandIndex = 0
        msgdlg.CancelCommandIndex = 1
        Dim res = Await msgdlg.ShowAsync()
        ListBox1.Items.Insert(0, "Gewählt: " + res.Id)
    End Sub
```

BEISPIEL 22.31: Auswahl per Methode auswerten

Die Methode zum Auswerten der Tasten:

```
    Private Sub CommandInvokedHandler(command As IUICommand)
        ListBox1.Items.Insert(0, "Taste [" + command.Label + "] wurde gedrückt!")
        ListBox1.Items.Insert(0, "ID ist [" + command.Id + "]")
    End Sub
...
```

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 22.3.3).

Ganz zum Schluss noch den einfachsten Aufruf der Methode:

```
...
    Private Sub Button_Click_6(sender As Object, e As RoutedEventArgs)
        Dim dlg = New MessageDialog("Hallo Welt!", "Mein Titel")
        dlg.ShowAsync()
    End Sub
...
```

HINWEIS: Dieser Aufruf ist asynchron, was bei einem modalen "Fenster" aber meist nicht relevant ist.

22.3.2 ContentDialog

Mit der Klasse *ContentDialog* steht Ihnen ein einfach zu konfigurierender Dialog für die alltäglichen Aufgaben zur Verfügung, der ohne aufwändige Ereignisbehandlung etc. auskommt. Sie können diesen modalen Dialog per Code oder XAML erzeugen, letzteres hat den Vorteil einer Entwurfsansicht im Designer. Der spätere Aufruf per *ShowAsync*-Methode ist in beiden Fällen gleich.

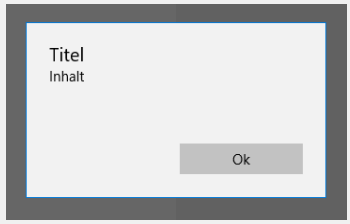
Doch zunächst ein simples Beispiel:

BEISPIEL 22.32: ***Erzeugen per Code***

VB

```
...
    Private Async Sub Button_Click_18(sender As Object, e As RoutedEventArgs)
        Dim cdlg = New ContentDialog() With {.Title = "Titel", .Content = "Inhalt",
                                             .PrimaryButtonText = "Ok"}
        Await cdlg.ShowAsync()
    End Sub
...
```

Ergebnis



Die XAML-Variante mit Auswertung der gedrückten Taste:

BEISPIEL 22.33: ***Erzeugen per XAML***

XAML

```
<Page
    x:Class="WinApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WinApp"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

        <ContentDialog x:Name="ContentDialog1"
           PrimaryButtonText="OK" IsPrimaryButtonEnabled="False"
           SecondaryButtonText="Abbruch" FullSizeDesired ="True"
           Opened="ContentDialog_Opened">
            <ContentDialog.TitleTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Image Source="ms-appx:///Assets/SmallLogo.png" Width="40"
                        Height="40" Margin="10,0"/>
                        <TextBlock Text="Beispiel für ContentDialog" />
                    </StackPanel>
                </DataTemplate>
            </ContentDialog.TitleTemplate>
```

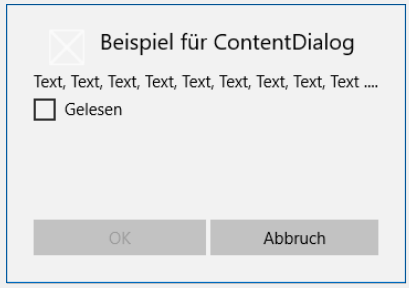
BEISPIEL 22.33: ***Erzeugen per XAML***

```
            <StackPanel>
                <TextBlock TextWrapping="WrapWholeWords">
                    Text, Text, Text, Text, Text, Text, Text, Text, Text ....
                </TextBlock>
                <CheckBox x:Name="CheckBox1" Content="Gelesen"/>
            </StackPanel>
        </ContentDialog>
...
```

VB

```
...
    Private Async Sub Button_Click_19(sender As Object, e As RoutedEventArgs)
        Dim result = Await ContentDialog1.ShowAsync()
        If result = ContentDialogResult.Primary Then

        Else

        End If
    End Sub
...
```

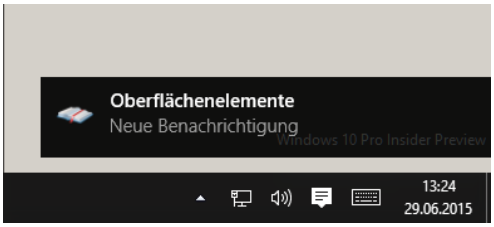
Ergebnis



22.3.3 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 wurde bereits in Windows 8 die sogenannten Popup-Benachrichtigungen oder "Toast notifications" implementiert. Diese werden durch das **System** entweder sofort oder zeitgesteuert am rechten Bildschirmrand (früher oben, jetzt unten) 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 VB-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

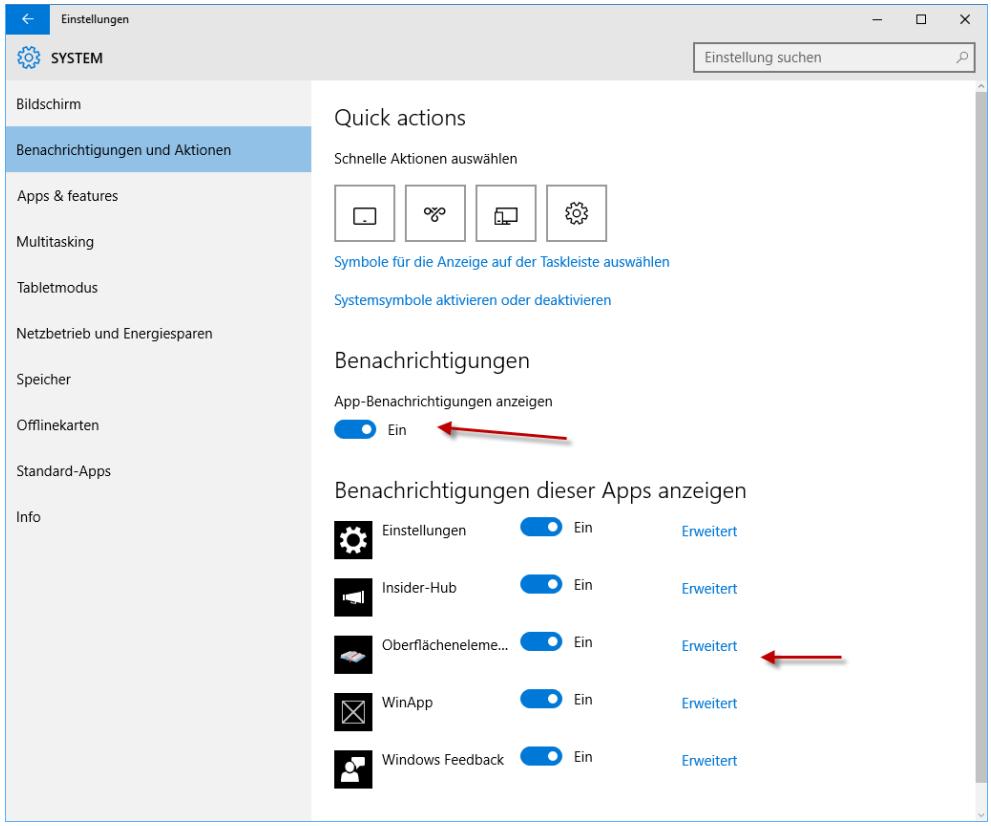
► 22.7.3 ToastNotifications einfach erzeugen

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.



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 22.34: XML-Code für eine Nachricht

XML

Eine Nachricht mit Kopfzeile, zwei Textzeilen sowie einem Bild:

```
<toast launch="Mein eigener Parameter">
  <visual version="1">
```

BEISPIEL 22.34: XML-Code für eine Nachricht

```
    <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

► 22.7.3 ToastNotifications einfach erzeugen

An dieser Stelle wollen wir uns nur mit der Verwendung anhand einiger Beispiele beschäftigen.

BEISPIEL 22.35: Einfache Meldung mit Kopfzeile und einer Textzeile anzeigen

VB

```
...
   Private Sub Button_Click_7(sender As Object, e As RoutedEventArgs)
        Dim tf As New DOKO.ToastNotificationFactory("Meine Kopfzeile",
                                                    "Meine wichtige Meldung an den User")
        tf.Show()
    End Sub
...
```

Ein Klick auf die Popup-Benachrichtigung zeigt im obigen Beispiel keine Reaktion. Dies wollen wir im nächsten Beispiel ändern.

BEISPIEL 22.36: Reaktionen auf Ereignisse implementieren

VB

In der *BasicPage1.xaml.vb*:

```
...
   Private Sub Button_Click_8(sender As Object, e As RoutedEventArgs)
```

Wir erzeugen eine dreizeilige Nachricht, die einen Parameter enthält:

```
        Dim tf As 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:

```
        Dim toast As ToastNotification = tf.CreateToast()
```

BEISPIEL 22.36: Reaktionen auf Ereignisse implementieren

```
        AddHandler toast.Activated, AddressOf toast_Activated
        AddHandler toast.Dismissed, AddressOf toast_Dismissed
```

Meldung über den *ToastNotificationManager* anzeigen:

```
        ToastNotificationManager.CreateToastNotifier().Show(toast)
    End Sub
...
```

Dieses Ereignis wird ausgelöst, wenn die Popup-Benachrichtigung **nicht** angeklickt wird:

```
    Private Async Sub toast_Dismissed(sender As ToastNotification,
                                      args As ToastDismissedEventArgs)
        Await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, Sub()
                          ListBox1.Items.Add("Toast Dismissed: " + args.Reason.ToString())
                      End Sub)
    End Sub
```

Dieses Ereignis wird ausgelöst, wenn die Popup-Benachrichtigung durch den Nutzer angeklickt wurde:

```
   Private Async Sub toast_Activated(sender As ToastNotification, args As Object)
        Await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, Sub()
                                                ListBox1.Items.Add("Toast Activated")
                                            End Sub)
    End Sub
```

Wird ein Parameter übergeben, rufen wir aus der *App.xaml.vb* diese Methode auf:

```
    Public Sub Parameterübergabe(args As String)
        ListBox1.Items.Add("Toast : " + args)
    End Sub
...
```

In der *App.xaml.vb*:

```
...
    Protected Overrides Sub OnLaunched(args As Activation.LaunchActivatedEventArgs)
        Dim rootFrame As Frame = Window.Current.Content
        If rootFrame Is Nothing Then
            rootFrame = New Frame()
            If args.PreviousExecutionState = ApplicationExecutionState.Terminated Then
            End If
            Window.Current.Content = rootFrame
        End If
        If rootFrame.Content Is Nothing Then
            If Not rootFrame.Navigate(GetType(BasicPage1), args.Arguments) Then
                Throw New Exception("Failed to create initial page")
            End If
        Else
```

BEISPIEL 22.36: Reaktionen auf Ereignisse implementieren

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:

```
            Dim page1 As BasicPage1 = CType(rootFrame.Content, BasicPage1)
            page1.Parameterübergabe(args.Arguments)
        End If
        Window.Current.Activate()
    End Sub
...
```

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 22.37: *ToastNotification* mit Bild

VB

```
...
    Private Sub Button_Click_9(sender As Object, e As RoutedEventArgs)
        Dim tf As 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()
    End Sub
...
```

Ergebnis

Last, but not least wollen wir Ihnen auch noch eine zeitgesteuerte Variante demonstrieren:

BEISPIEL 22.38: Zeitgesteuerte Anzeige der Benachrichtigung

VB

```
...
    Private Sub Button_Click_10(sender As Object, e As RoutedEventArgs)
```

BEISPIEL 22.38: Zeitgesteuerte Anzeige der Benachrichtigung

```
        Dim tf As 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))
    End Sub
...
```

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.

22.3.4 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.

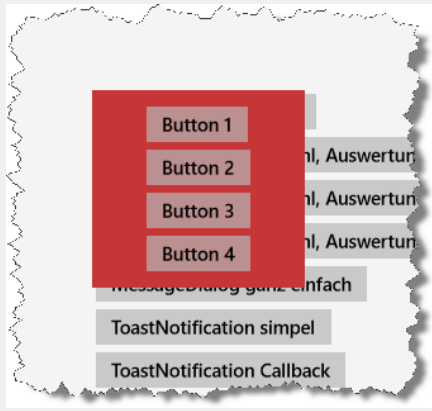
- per VB-Code
  Hier ist es günstig, wenn Sie den Content des Popup-Controls zum Beispiel als UserControl erstellen (XAML-Code) und dann als Child zuweisen. Andernfalls haben Sie schnell sehr viel Code produziert, nur um eine Schaltfläche und ein paar andere Controls anzuzeigen.

BEISPIEL 22.39: PopUp per XAML-Code erzeugen

XAML

```
...
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition></ColumnDefinition>
                    <ColumnDefinition></ColumnDefinition>
                </Grid.ColumnDefinitions>
...
                <Popup x:Name="Popup1" IsLightDismissEnabled="True" >
                    <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



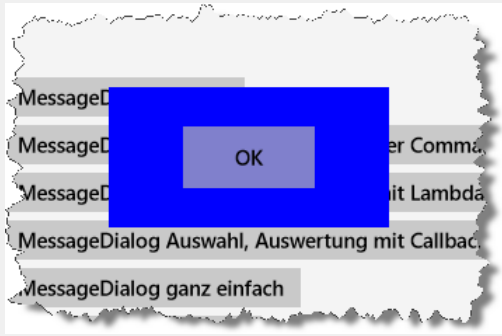
Etwas umständlicher wird es per VB-Code gerade dann, wenn Sie die enthaltenen Controls parametrieren wollen:

BEISPIEL 22.40: PopUp per VB-Code erzeugen

VB

```
...
Imports Windows.UI.Popup
...
   Private Sub Button_Click_13(sender As Object, e As RoutedEventArgs)
        Dim popup As New Popup()
        popup.Width = 200
        popup.Height = 150
        popup.VerticalOffset = 200
        popup.HorizontalOffset = 200
        Dim bord As New Border() With { _
             .Width = 200, _
             .Height = 100, _
            .Background = New SolidColorBrush(Colors.Blue) _
        }
        bord.Child = New Button() With { _
             .Width = 100, _
             .Height = 50, _
             .HorizontalAlignment = HorizontalAlignment.Center, _
             .Content = "OK" _
        }
        popup.Child = bord
        popup.IsLightDismissEnabled = True
        popup.IsOpen = True
    End Sub
...
```

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 *IsLight-DismissEnabled.* 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 22.41: UserControls im Popup einblenden und positionieren

VB

```
...
  Public Shared Function ShowPopup(source As FrameworkElement,
        control As UserControl, Optional IsLightDismissEnabled As Boolean = True) As Popup
        Dim flyout As New Popup()
        flyout.IsLightDismissEnabled = IsLightDismissEnabled
        Dim gt As GeneralTransform = source.TransformToVisual(Window.Current.Content)
        Dim 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
        Dim transitions = New TransitionCollection()
        transitions.Add(New PopupThemeTransition() With { _
             .FromHorizontalOffset = 0,
             .FromVerticalOffset = 100 _
        })
        flyout.ChildTransitions = transitions
        flyout.IsOpen = True
        Return flyout
    End Function
...
```

Die Verwendung:

```
     Private Sub Button_Click_12(sender As Object, e As RoutedEventArgs)
        ShowPopup(Button12, New MyUserControl1())
    End Sub
...
```

Der obigen Funktion übergeben Sie ein Bezugs-Control (z.B. eine Schaltfläche) und ein *User-Control* (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

[1] Quelle: *http //blogs.msdn.com/b/eternalcoding/archive/2012/07/03/tips-and-tricks-for-c-metro-developers-the-flyout-control.aspx*

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 22.42: 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>
        </Popup>
...
```

VB

Beim Einblenden müssen wir das Popup am rechten Rand ausrichten:

```
    Private Sub Button_Click_14(sender As Object, e As RoutedEventArgs)
        Popup2.HorizontalOffset = Window.Current.Bounds.Width - 400
        Popup2.IsOpen = True
    End Sub
```

22.3.5 Das PopupMenu einsetzen

Ein naher, etwas spezialisierter Verwandter des Popups ist das *PopupMenu*. Diesem können Sie einzelne *UICommand*-Objekte hinzufügen (siehe auch 22.3.1, Seite 1115), 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 22.43: Verwendung *PopupMenu*

VB

```
...
Imports Windows.UI.Popups
...
```

Mit dem Klick auf eine Schaltfläche erstellen wir das *PopupMenu* und zeigen es an:

```
    Private Async Sub Button_Click_17(sender As Object, e As RoutedEventArgs)
        Dim popmenu As New PopupMenu()
```

Der erste Eintrag nutzt eine Lambda-Funktion als Reaktion auf den Klick:

```
        popmenu.Commands.Add(New UICommand("Erster Eintrag", Sub(command)
                                        ListBox1.Items.Add(command.Label)
                                    End Sub))
```

Die folgenden Einträge sind mit einem extra Handler verknüpft:

```
        popmenu.Commands.Add(New UICommand("Zweiter Eintrag",
                        New UICommandInvokedHandler(AddressOf test), "2"))
        popmenu.Commands.Add(New UICommand("Dritter Eintrag",
                        New UICommandInvokedHandler(AddressOf test), "dritter"))
        popmenu.Commands.Add(New UICommand("Vierter Eintrag",
                        New UICommandInvokedHandler(AddressOf test), "vierter"))
```

Aufrufvariante 1:

```
        Dim transform As GeneralTransform = TryCast(sender, Button).TransformToVisual(Me)
        Dim point As Point = transform.TransformPoint(New Point(100, 150))
        Await popmenu.ShowAsync(point)
```

Aufrufvariante 2:

```
        Dim transform As GeneralTransform = TryCast(sender, Button).TransformToVisual(Me)
        Dim point As Point = transform.TransformPoint(New Point(100, 150))
        Await popmenu.ShowForSelectionAsync(New Rect(point,
                        New Size(CType(sender, Button).ActualWidth,
                        CType(sender, Button).ActualHeight)))
```

Aufrufvariante 3 (unsere Erweiterungsmethode):

```
          Await popmenu.ShowAsync(CType(sender, Button))
    End Sub
```

Welche Aufrufvariante intuitiver ist, dürfte schnell erkennbar sein.

VB

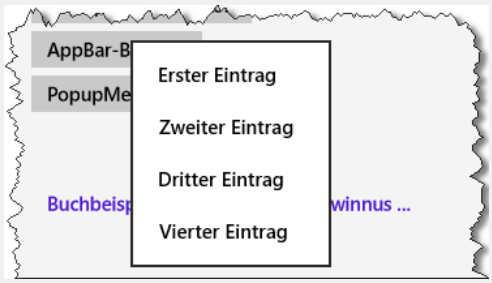
Mit der folgenden Erweiterungsmethode wird das *PopupMenu* an einem Control ausgerichtet und angezeigt:

```
Imports Windows.UI.Popups
Module Erweiterungsmethoden
    <System.Runtime.CompilerServices.Extension> _
    Public Async Function ShowAsync(menu As PopupMenu, control As FrameworkElement) _
                                As Task(Of IUICommand)
```

BEISPIEL 22.43: Verwendung *PopupMenu*

```
        Dim transform As GeneralTransform = _
                  control.TransformToVisual(TryCast(Window.Current.Content, Frame))
        Dim point As Point = transform.TransformPoint(New Point(100, 150))
        Return Await menu.ShowForSelectionAsync(New Rect(point,
                  New Size(control.ActualWidth, control.ActualHeight)))
    End Function
End Module
...
```

Ergebnis



22.3.6 Eine AppBar verwenden

Wir hatten uns ja bereits im Rahmen der Oberflächengestaltung (Abschnitt 19.1.3, Seite 957) 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 die Klasse *CommandBar* werfen, es handelt sich um eine recht spezialisierten *AppBar*, der sich auf Buttons und ToggleButtons beschränkt, diese Einschränkung aber mit einer selten simplen Umsetzung belohnt.

BEISPIEL 22.44: *AppBar* verwenden

XAML

```
...
    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" 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"/>
```

BEISPIEL 22.44: *AppBar* verwenden

```
                        <TextBlock>Rubrik B</TextBlock>
                    </StackPanel>
                </Button>
            </StackPanel>
        </Grid>
    </AppBar>
</Page.TopAppBar>
```

Die Befehlsleiste am unteren Rand implementieren wir als *CommandBar*, die wesentlich einfachere Programmierung dürfte schnell ersichtlich sein:

```
<Page.BottomAppBar>
    <CommandBar IsOpen="True">
```

Drei Schaltflächen, die Icons können wir aus dem vorhandene Fundus abrufen:

```
        <AppBarButton Icon="Edit" Label="Edit" Click="Button_Click_3"/>
        <AppBarButton Icon="Save" Label="Save" Click="Button_Click_4"/>
        <AppBarButton Icon="Delete" Label="Delete"/>
```

Für untergeordnete Aufgaben bieten sich die sogenannte *SecondaryCommands* an, diese werden über die Schaltfläche mit den drei Punkten eingeblendet:

```
        <CommandBar.SecondaryCommands>
            <AppBarButton Icon="Help" Label="Help"/>
        </CommandBar.SecondaryCommands>
    </CommandBar>
</Page.BottomAppBar>
...
```

VB

Beim Klick auf die "Editieren"-Schaltfläche wird ein *Popup* geöffnet, Inhalt ist ein *UserControl* (quasi der Ersatz für einen Dialog):

```
...
  Private Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
      Dim flyout As New Popup()
      flyout.IsLightDismissEnabled = True
      Dim uc As UserControl = New MyUserControl1()
      flyout.Child = uc
```

Wir richten den Dialog an der linken Kante des Buttons aus, die Höhe müssen wir leider direkt setzen:

```
      Dim gt As GeneralTransform = TryCast(sender,
                                    Button).TransformToVisual(Window.Current.Content)
      Dim absolutePosition = gt.TransformPoint(New Point(0, 0))
      uc.Measure(New Size(Double.PositiveInfinity, Double.PositiveInfinity))
      flyout.VerticalOffset = absolutePosition.Y - 350
      flyout.HorizontalOffset = absolutePosition.X
      Dim transitions = New TransitionCollection()
      transitions.Add(New PopupThemeTransition() With { _
```

BEISPIEL 22.44: *AppBar* verwenden

```
            .FromHorizontalOffset = 0,
            .FromVerticalOffset = 100 _
        })
        flyout.ChildTransitions = transitions
        flyout.IsOpen = True
    End Sub
```

Der Klick auf die Schaltfläche "Sichern" zaubert ein *PopupMenu* hervor:

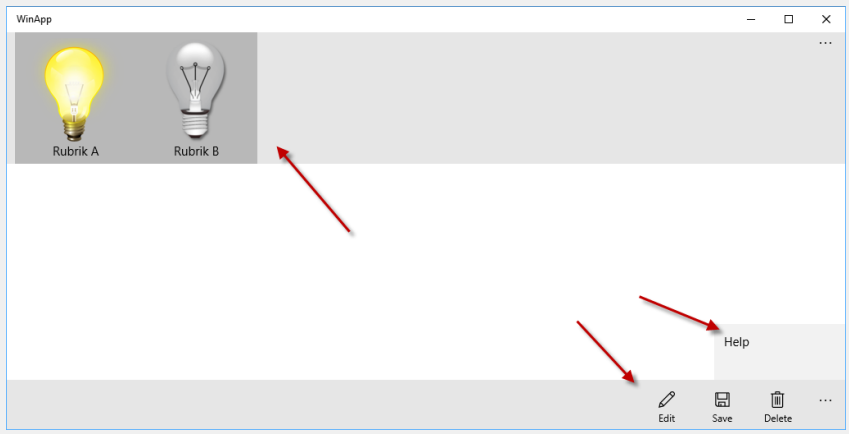
```
    Private Async Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        Dim popupMenu = New PopupMenu()
        popupMenu.Commands.Add(New UICommand("Eintrag 1"))
        popupMenu.Commands.Add(New UICommand("Eintrag 2"))
        popupMenu.Commands.Add(New UICommand("Eintrag 4"))
        popupMenu.Commands.Add(New UICommand("Eintrag 5"))
        Dim transform As GeneralTransform = TryCast(sender, Button).TransformToVisual(Me)
        Dim point As Point = transform.TransformPoint(New Point(100, 150))
```

Hier nutzen wir wieder unsere Erweiterungsmethode zur Anzeige:

```
        Await popupMenu.ShowAsync(point)
    End Sub
...
```

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.

22.4 Datenbanken und Windows Store Apps

Oh, da scheinen die Entwickler immer noch etwas 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[1].

22.4.1 Der Retter in der Not: SQLite!

Auf der Website

LINK: http://timheuer.com/blog/default.aspx

wird eine Lösung vorgestellt, die allerdings nicht von Microsoft stammt.

Wir hatten uns in unserem letzten Datenbankbuch bereits ausführlicher[2] damit beschäftigt: Es geht um SQLite, das mittlerweile auch in einer für WinRT-tauglichen Form verfügbar ist.

22.4.2 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 als PDF auslagern) und so wollen wir uns auf einige wichtige Punkte beschränken.

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] Das war schon in der Vorgänger-Version so.

[2] Unter diesen Umständen werden wir das entsprechende Kapitel in unserem Datenbankbuch wohl noch kräftig ausbauen müssen.

[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.

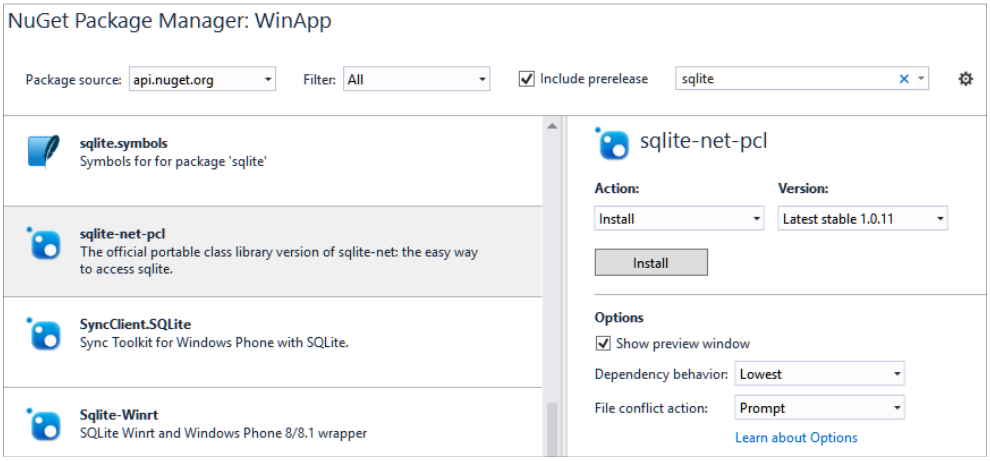
Für alle, die sich jetzt voller Euphorie auf die SQLite-Entwicklung stürzen wollen, gleich ein beachtlicher Dämpfer:

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!

22.4.3 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.

Die Installation ist mittlerweile recht komfortabel geworden, nutzen Sie einfach NuGet, um das Paket *sqlite-net-pcl* herunterzuladen und zu installieren.



Zum Zeitpunkt der Drucklegung gab es noch Probleme im Zusammenhang mit dem Entwurf von Windows 10 Apps, die DLL *sqlite3.dll* im Package-Verzeichnis

```
\\...\packages\SQLitePCL.raw_basic.0.7.1\build\native\sqlite3_dynamic\winrt81\x86
```

musste noch durch eine andere Version ersetzt werden, die Sie hier herunterladen können:

LINK: `http://www.sqlite.org/2015/sqlite-dll-win32-x86-3081002.zip`

22.4.4 Wie kommen wir zu einer neuen Datenbank?

Drei grundsätzliche Varianten bieten sich bei der vorliegenden SQLite-Lösung an:

- Sie erstellen die Datenbank per VB-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 22.45: Erstellen der Datenbank aus VB-Klassen zur Laufzeit

VB

```
...
Imports SQLite
Imports Windows.Storage
Imports System.Threading.Tasks
...
Public NotInheritable Class BasicPage1
    Inherits Common.LayoutAwarePage
```

Unsere Verbindung zur Datenbank:

```
    Private db As SQLiteConnection
    Private artikeltabelle As TableQuery(Of Artikel)

    Public Sub New()
        Me.InitializeComponent()
        AddHandler Loaded, AddressOf BasicPage1_Loaded
    End Sub
```

Mit dem Laden der Seite wird auch die Datenbank im App-Datenverzeichnis erstellt (ist diese bereits vorhanden, wird sie automatisch geöffnet):

```
    Private Async Sub BasicPage1_Loaded(sender As Object, e As RoutedEventArgs)
       db = New SQLite.SQLiteConnection(
                   Path.Combine(Windows.Storage.ApplicationData.Current.LocalFolder.Path,
                   "firma.db"))
```

Erstellen einer Tabelle:

```
        db.CreateTable(Of 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.

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

BEISPIEL 22.45: Erstellen der Datenbank aus VB-Klassen zur Laufzeit

Spalten haben sollen:

```
Imports SQLite
...
```

Unsere Beispiel-Klasse *Artikel*:

```
Public Class Artikel
    Implements INotifyPropertyChanged

    Public Sub New()
    End Sub
```

Ein Konstruktor zum einfachen Erstellen neuer Instanzen:

```
    Public Sub New(bezeichnung As String, preis As Double)
        Me._bezeichnung = bezeichnung
        Me._preis = CSng(preis)
        Me._anzahl = 0
        Me._ausverkauft = True
    End Sub
```

Unser Primärschlüssel (ein Zählerwert):

```
    Private _id As Integer
    <SQLite.AutoIncrement, PrimaryKey> _
    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(value As Integer)
            _id = value
            Me.NotifyPropertyChanged("Id")
        End Set
    End Property
```

Ein indiziertes Textfeld in der Datenbank mit der Länge 100:

```
    Private _bezeichnung As String
    <Indexed, MaxLength(100)> _
    Public Property Bezeichnung() As String
        Get
            Return _bezeichnung
        End Get
        Set(value As String)
            _bezeichnung = value
            Me.NotifyPropertyChanged("Bezeichnung")
        End Set
    End Property
```

BEISPIEL 22.45: Erstellen der Datenbank aus VB-Klassen zur Laufzeit

Ein *Single*-Feld:

```
Private _preis As Single
Public Property Preis() As Single
    Get
        Return _preis
    End Get
    Set(value As Single)
        _preis = value
        Me.NotifyPropertyChanged("Preis")
    End Set
End Property
```

Ein *Integer*-Feld:

```
Private _anzahl As Integer
Public Property Anzahl() As Integer
    Get
        Return _anzahl
    End Get
    Set(value As Integer)
        _anzahl = value
        Me.NotifyPropertyChanged("Anzahl")
    End Set
End Property
```

Ein *Boolean*-Feld:

```
Public Property Ausverkauft() As Boolean
    Get
        Return _ausverkauft
    End Get
    Set(value As Boolean)
        _ausverkauft = value
        Me.NotifyPropertyChanged("Ausverkauft")
    End Set
End Property
```

Dieses Feld wird nur in der App benötigt, es gibt keine Entsprechung in der Datenbank:

```
<Ignore> _
Public ReadOnly Property ID_Bezeichnung() As String
    Get
        Return _id.ToString() & "_" & Bezeichnung
    End Get
End Property
```

Für die Anzeige im Listenfeld eine überschriebene *ToString*-Methode:

```
Public Overrides Function ToString() As String
```

BEISPIEL 22.45: Erstellen der Datenbank aus VB-Klassen zur Laufzeit

```
        Return String.Format("Id:{0} Bez:{1} Preis:{2}", _id, _bezeichnung, _preis)
    End Function
...
```

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.

HINWEIS: Mit dem SQLite-Administrator (siehe *http://sqliteadmin.orbmu2k.de/*) können Sie die Datenbanken komfortabel entwerfen.

22.4.5 Wie werden die Daten manipuliert?

An dieser Stelle halten wir uns mit weitschweifigen Ausführungen zurück, ein kleines Beispiel soll genügen.

BEISPIEL 22.46: Manipulieren der SQLite-Datenbank

VB

```
...
    Private Async Sub BasicPage1_Loaded(sender As Object, e As RoutedEventArgs)
...
        db = New SQLite.SQLiteConnection(Path.Combine( _
                Windows.Storage.ApplicationData.Current.LocalFolder.Path, "firma.db"))
        db.CreateTable(Of Artikel)()
```

Wir fragen die Anzahl der Datensätze in der Tabelle *Artikel* ab:

```
        Dim anzahl As Integer = db.ExecuteScalar(Of Integer)("SELECT COUNT(*) FROM artikel")
```

Sind keine Datensätze vorhanden, erstellen wir vier neue Einträge:

```
        If anzahl = 0 Then
            db.RunInTransaction(Sub()
                                    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))
                                End Sub)
        End If
```

Wir fragen die komplette Tabelle ab und zeigen diese an:

```
        artikeltabelle = db.Table(Of Artikel)()
        ListBox1.ItemsSource = artikeltabelle
    End Sub
...
```

Einen einzelnen Artikel einfügen und die Tabelle erneut abfragen:

```
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
```

Artikel erzeugen:

```
        Dim art As 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(Of Artikel)()
        ListBox1.ItemsSource = artikeltabelle
    End Sub
```

BEISPIEL 22.46: Manipulieren der SQLite-Datenbank

Wir löschen den ersten Artikel in der Liste:

```
    Private Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        db.Delete(artikeltabelle.First())
```

Tabelle erneut abfragen und anzeigen:

```
        artikeltabelle = db.Table(Of Artikel)()
        ListBox1.ItemsSource = artikeltabelle
    End Sub
```

Wir fragen alle Artikel ab, die einen Preis größer 1,4 Euro haben:

```
    Private Sub Button_Click_4(sender As Object, e As RoutedEventArgs)
        ListBox1.ItemsSource = db.Query(Of Artikel)( _
                               "SELECT * FROM artikel WHERE preis > 1.4")
    End Sub
...
```

22.5 Vertrieb der App

Kommen wir noch kurz zum Thema "Distribution von Windows 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.

22.5.1 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.

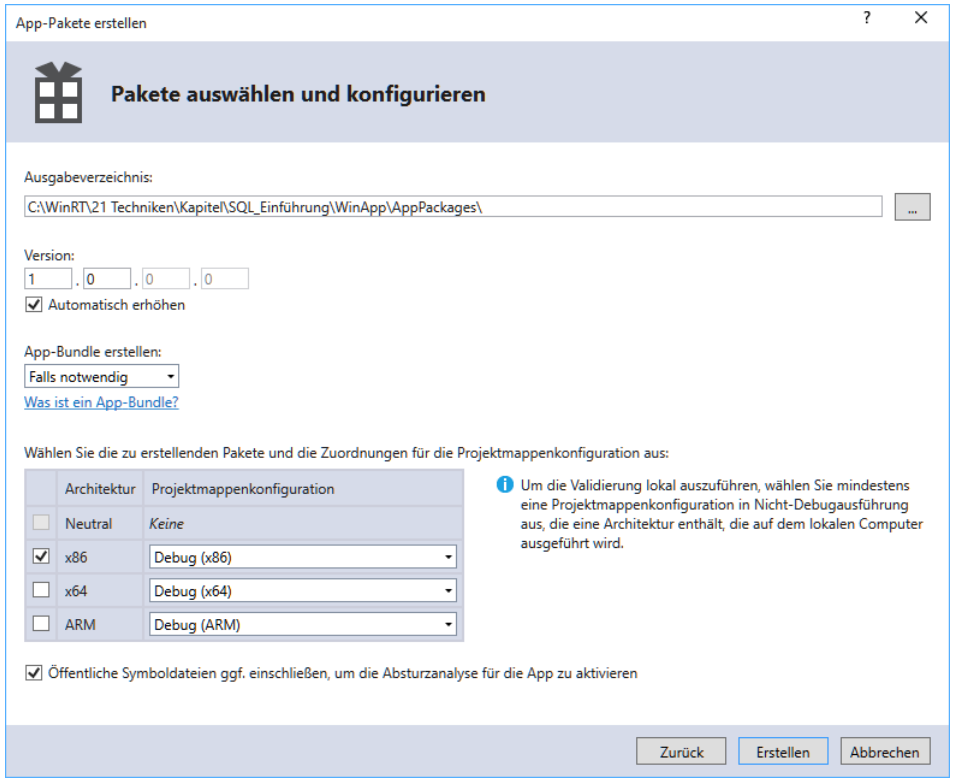
Sie könnten natürlich alle Schritte zum fertigen Package auch an der Kommandozeile realisieren (VB-/C#-Compiler, *MakeAppx.exe*, *Signtool.exe*), wir gehen jedoch ganz einfach mal davon aus, dass Sie mit Visual Studio arbeiten und sich so eine Menge Arbeit ersparen.

[1] So weit man dies bei Software überhaupt sagen kann.

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:

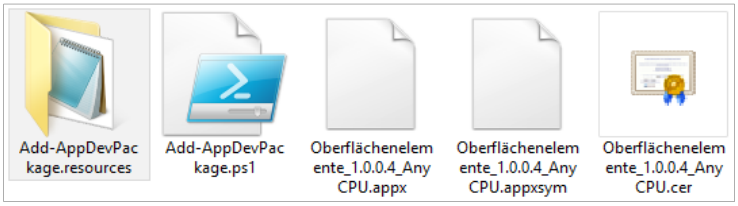
Hier haben Sie die Auswahl, ob Sie einen Paket für den Shop oder für eine lokale Distribution erstellen wollen. Wählen Sie zunächst die zweite Variante. Nachfolgend wählen Sie das Ausgabeverzeichnis, die Versionsnummer und die Architektur aus:



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.

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*)

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.

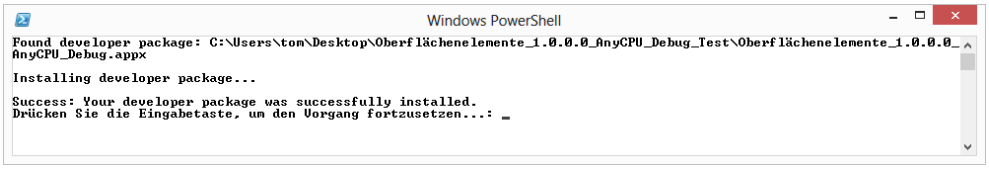
22.5.2 App-Installation per Skript

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 freige-

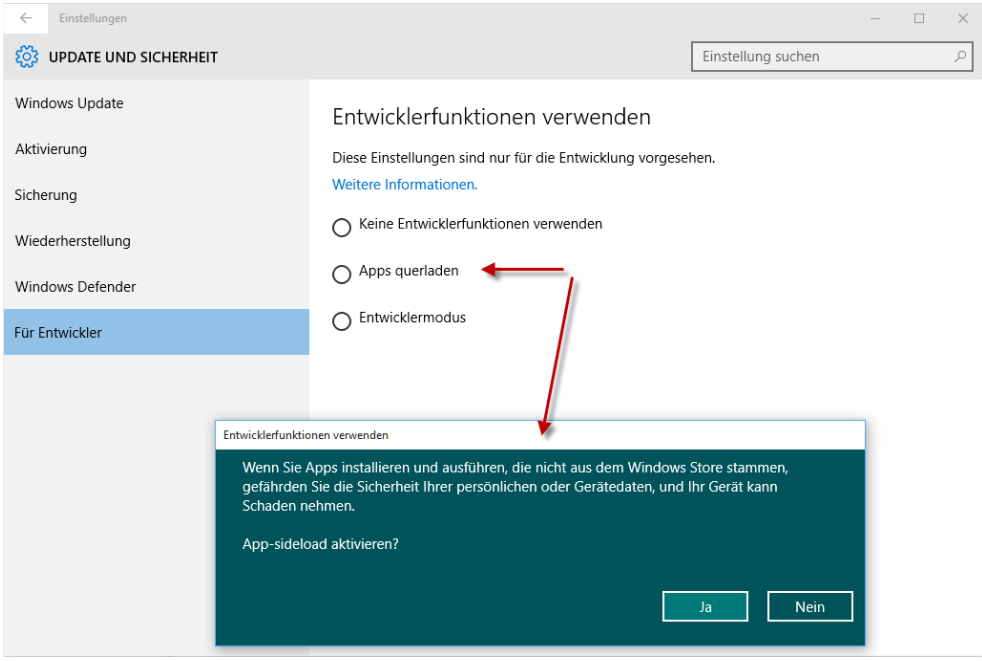
schalteten PC, der einer Domain zugeordnet ist. Andere Windows-Versionen müssen mit einem speziellen Key für die Installation freigeschaltet werden.

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.

Neu in Windows 10 ist die Möglichkeit, dass der Anwender die Installation ohne den Windows Store erlaubt:



Es ist allerdings fraglich, ob ein Anwender diese Option freigibt und wie Microsoft diese Option in Zukunft unterstützt.

22.6 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.

22.6.1 Quellcodes im Installationsverzeichnis

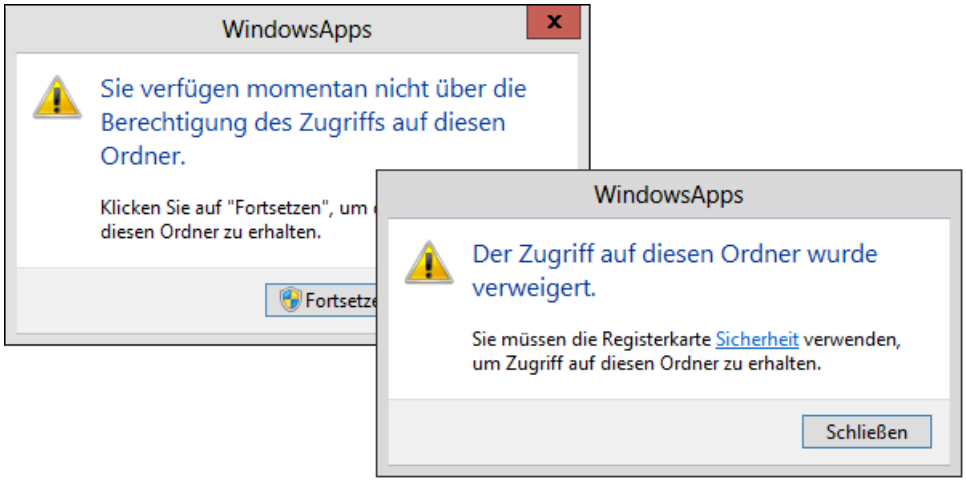
Haben Sie schon .NET-Apps entwickelt, ist Ihnen eine Schwachstelle von .NET-Programmen sicher geläufig: Mit geeigneten Tools (.NET-Reflector, ILSpy etc.) können Sie problemlos einen Blick in die Arbeit anderer Programmierer werfen. Sie haben die freie Wahl der Programmiersprache, lediglich die Variablennamen 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.

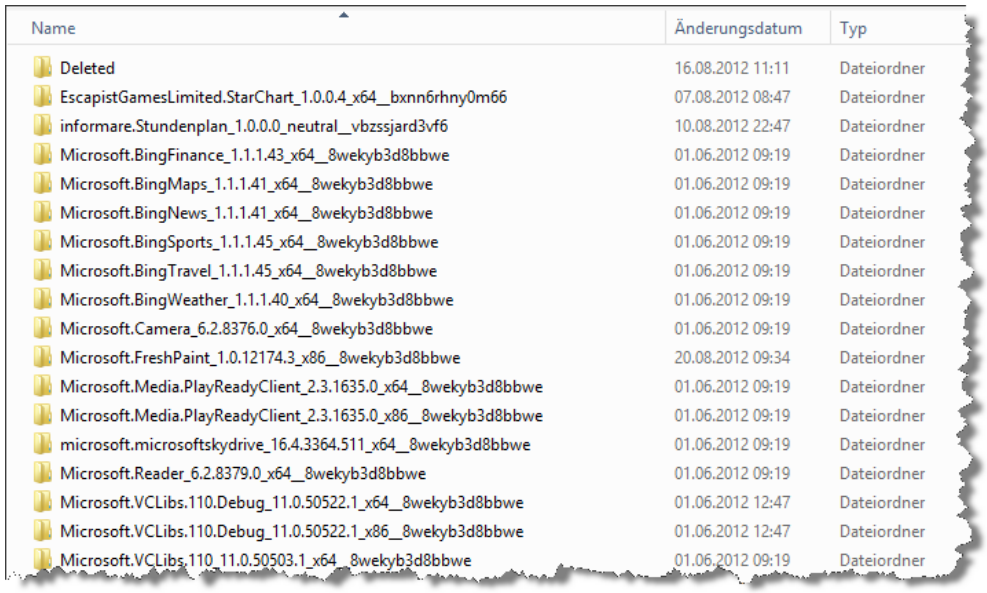
Folgende Meldung begrüßt Sie:



Das sieht zunächst recht gut aus, doch lassen Sie sich von dem Geplänkel nicht abschrecken:

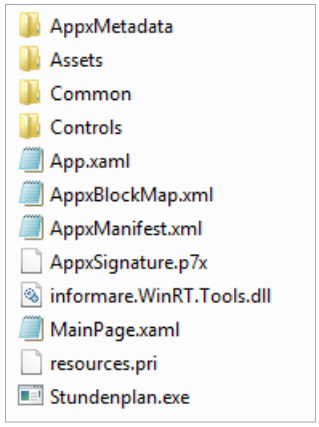
- Setzen Sie die Gruppe *Administratoren* als Besitzer des Ordners ein[1].
- Übernehmen Sie alle Rechte für die Gruppe *Administratoren* (Vollzugriff).
- Greifen Sie auf das Verzeichnis mit vollen Rechten zu.

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:



- AppxMetadata
- Assets
- Common
- Controls
- App.xaml
- AppxBlockMap.xml
- AppxManifest.xml
- AppxSignature.p7x
- informare.WinRT.Tools.dll
- MainPage.xaml
- resources.pri
- Stundenplan.exe

[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.

22.6.2 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.

22.7 Praxisbeispiele

22.7.1 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.vb* und nehmen Sie folgende Erweiterungen vor:

```
...
Imports Windows.Storage
Imports Windows.Storage.Search
Imports Windows.UI.Xaml.Navigation

Public NotInheritable Class BasicPage1
    Inherits Page
```

Eine Instanz unserer *StorageFileQuery*:

```
    Private query As StorageFileQueryResult
```

Mit dem Öffnend der Seite erzeugen wir zunächst eine Dateilistenabfrage für das App-Datenverzeichnis:

```
    Protected Overrides Async Sub OnNavigatedTo(e As NavigationEventArgs)
        Dim queryOptions As New QueryOptions(CommonFileQuery.DefaultQuery, Nothing)
```

Mit Unterverzeichnissen:

```
        queryOptions.FolderDepth = FolderDepth.Deep
        queryOptions.IndexerOption = IndexerOption.UseIndexerWhenAvailable
```

Abfrage erzeugen:

```
        query = ApplicationData.Current.LocalFolder.CreateFileQueryWithOptions(queryOptions)
```

Ereignishandler registrieren:

```
        AddHandler query.ContentsChanged, AddressOf 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)
    End Sub
```

Unser Ereignishandler (wir müssen auf den Vordergrundthread umschalten, das Ereignis wird extern ausgelöst):

```
    Private Async Sub query_ContentsChanged(sender As IStorageQueryResultBase, args As Object)
        Await Dispatcher.RunAsync(Windows.UI.Core.CoreDispatcherPriority.Normal, Async Sub()
```

Wir rufen die neue Dateiliste ab:

```
                                            ListBox1.Items.Clear()
                                            For Each f In Await query.GetFilesAsync()
                                               ListBox1.Items.Add("Änderung: " + f.Path)
                                            Next
                                        End Sub)
    End Sub
```

Ein Verzeichnis testweise erzeugen:

```
    Private Async Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Dim folder As StorageFolder = _
              Await ApplicationData.Current.LocalFolder.CreateFolderAsync("Testordner",
              CreationCollisionOption.GenerateUniqueName)
    End Sub
```

Eine Datei testweise erzeugen:

```
   Private Async Sub Button_Click_3(sender As Object, e As RoutedEventArgs)
        Dim file As StorageFile = Await _
                    ApplicationData.Current.LocalFolder.CreateFileAsync("Hallo.txt",
                    CreationCollisionOption.GenerateUniqueName)
    End Sub
...
End Class
```

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).

22.7.2 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:

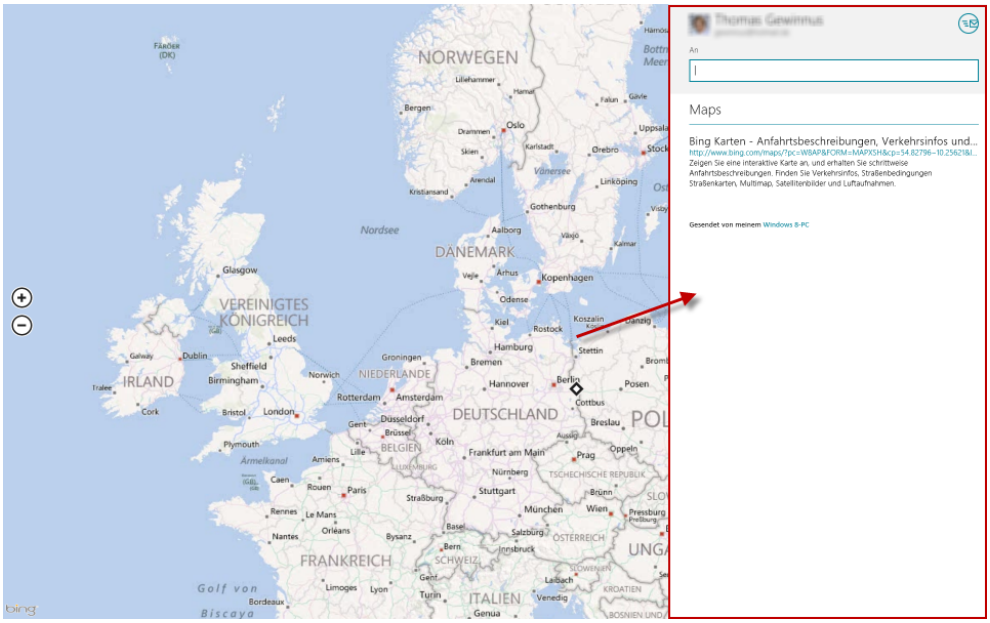
```
<ScrollViewer VerticalScrollBarVisibility="Auto" HorizontalScrollBarVisibility="Auto"
        Margin="50" Grid.Row="0" ZoomMode="Disabled">

    <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:

```
<Applications>
  <Application Id="App"
    Executable="$targetnametoken$.exe"
    EntryPoint="WinApp_ZielFreigabe.App">
    <uap:VisualElements
      DisplayName="WinApp_ZielFreigabe"
      Square150x150Logo="Assets\Logo.png"
      Square44x44Logo="Assets\SmallLogo.png"
      Description="WinApp_ZielFreigabe"
      BackgroundColor="#464646">
      <uap:SplashScreen Image="Assets\SplashScreen.png" />
    </uap:VisualElements>
    <Extensions>
      <uap:Extension Category="windows.shareTarget">
        <uap:ShareTarget>
          <uap:SupportedFileTypes>
            <uap:SupportsAnyFileType />
          </uap:SupportedFileTypes>
          <uap:DataFormat>Text</uap:DataFormat>
          <uap:DataFormat>Bitmap</uap:DataFormat>
          <uap:DataFormat>StorageItems</uap:DataFormat>
        </uap:ShareTarget>
      </uap:Extension>
```

```
    </Extensions>
    </Application>
  </Applications>
```

Ihre App wird später nur dann 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.

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.vb)

Wir implementieren zunächst den Eintrittspunkt *OnShareTargetActivated*. Öffnen Sie die Datei *App.xaml.vb* und fügen Sie folgende Methode hinzu:

```
    Protected Overrides Sub OnShareTargetActivated(args As ShareTargetActivatedEventArgs)
        Dim rootFrame = New Frame()
        rootFrame.Navigate(GetType(InsertPage), args.ShareOperation)
        Window.Current.Content = rootFrame
        Window.Current.Activate()
    End Sub
```

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.vb)

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:

```
...
Imports Windows.ApplicationModel.DataTransfer
Imports Windows.ApplicationModel.DataTransfer.ShareTarget
Imports Windows.Storage.Streams
Imports Windows.UI.Xaml.Media.Imaging
Imports Windows.UI.Xaml.Navigation

Public NotInheritable Class InsertPage
    Inherits Common.LayoutAwarePage
```

Das folgende Objekt nimmt das an die Seite übergebene *ShareOperation*-Objekt auf:

```
    Private shareop As ShareOperation
```

Mit dem Öffnen der Seite rufen wir zunächst den Übergabeparameter ab und speichern diesen:

```
Protected Overrides Async Sub OnNavigatedTo(e As NavigationEventArgs)
    shareop = DirectCast(e.Parameter, ShareOperation)
```

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) Then
        Try
```

Text anzeigen:

```
            ListBox1.Items.Add("Text = " + Await shareop.Data.GetTextAsync())
        Catch ex As Exception
            ListBox1.Items.Add("Error: " + ex.Message)
        End Try
    End If
```

Auswerten, ob eine Dateiliste übergeben wurde:

```
    If shareop.Data.Contains(StandardDataFormats.StorageItems) Then
        ListBox1.Items.Add("Dateien: ")
        Try
```

Die Dateiliste durchlaufen und die kompletten Dateinamen anzeigen:

```
            Dim items = Await shareop.Data.GetStorageItemsAsync()
            For Each item In items
                ListBox1.Items.Add(item.Path)
            Next
        Catch ex As Exception
            ListBox1.Items.Add("Error: " + ex.Message)
        End Try
    End If
```

Wie schon bei der Zwischenablage macht uns auch hier die Bitmap die meiste Arbeit:

```
   If shareop.Data.Contains(StandardDataFormats.Bitmap) Then
        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:

```
    If shareop.Data.Contains(StandardDataFormats.Bitmap) Then
        ListBox1.Items.Add("Bitmap: ")
```

```
            Try
                Dim imgdata As IRandomAccessStreamReference = _
                              Await shareop.Data.GetBitmapAsync()
                If imgdata IsNot Nothing Then
                    Dim bi As New BitmapImage()
                    bi.SetSource(Await imgdata.OpenReadAsync())
                    Image1.Source = bi
                End If
            Catch ex As Exception
                ListBox1.Items.Add("Error: " + ex.Message)
            End Try
        End If
```

Hier können wir dem System die erfolgreiche Datenübernahme mitteilen:

```
        shareop.ReportDataRetrieved()
    End Sub
```

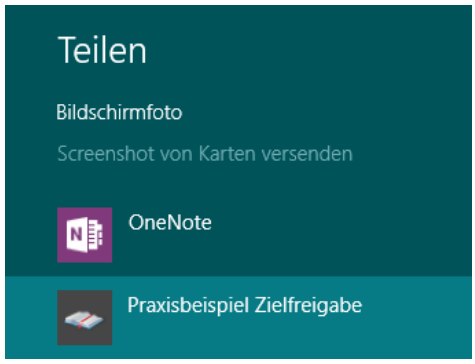
Im Normalfall wird der Nutzer eine Aktion in diesem Fenster auslösen (Email versenden etc.), wir belassen es beim Ändern der Schaltflächenbeschriftung. Mit dem Aufruf *ReportCompleted* wird die Seite geschlossen, die Share-Operation ist abgeschlossen:

```
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Button2.Content = "Verarbeitet ..."
        shareop.ReportCompleted()
    End Sub
End Class
```

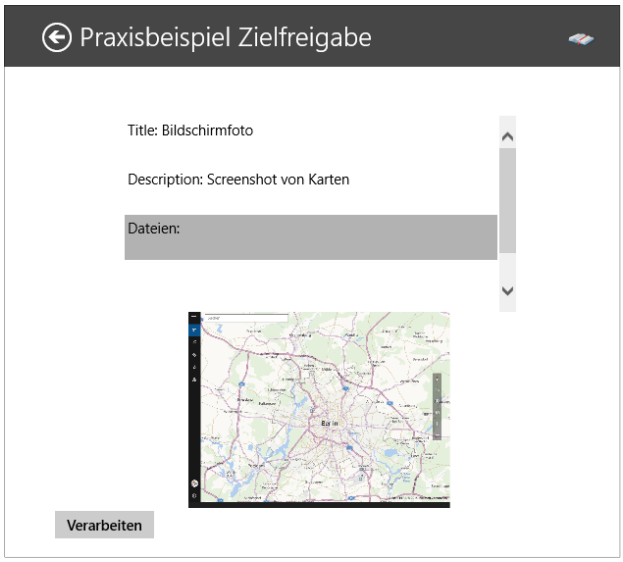
Test

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 22.2.2).

Rufen Sie die Funktion *Teilen* auf (*Win+H*), 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.

22.7.3 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 22.3.3 ab Seite 1120 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

```
Imports System.Linq
Imports Windows.Data.Xml.Dom
Imports Windows.UI.Notifications

Namespace DOKO
    Class ToastNotificationFactory
```

Interne Instanz der *ToastNotification* (für die Methode *Hide*) speichern:

```
        Private _toast As ToastNotification
```

Unsere Schnittstelle nach außen:

```
        Public Property Kopfzeile() As String
...
        Public Property Textzeile1() As String
...
        Public Property Textzeile2() As String
...
        Public Property Bild() As String
...
        Public Property Parameter() As String
...
        Public Property LongRunning() As Boolean
...
        Public Property SoundLoop() As Boolean
...
        Private m_SoundLoop As Boolean
...
        Public Property Sound() As String
...
```

HINWEIS: Alle Eigenschaften sind optional bzw. werden im Konstruktor initialisiert. Sie müssen mindestens eine Textzeile parametrieren. Die weiteren möglichen Varianten entnehmen Sie dem Abschnitt 22.3.3.

Der Standardkonstruktor zum Initialisieren der Werte:

```
        Public Sub New()
            Kopfzeile = ""
            Textzeile1 = ""
            Textzeile2 = ""
            Bild = ""
            Parameter = ""
            LongRunning = False
```

```
            SoundLoop = False
            Silent = False
            Sound = "ms-winsoundevent:Notification.Looping.Alarm"
        End Sub
```

Eine Überladung des Konstruktors mit optionalen Parametern erlaubt eine schnellere Definition der *ToastNotification*:

```
        Public Sub New(kopfzeile__1 As String, Optional textzeile1__2 As String = "",
                                    Optional textzeile2__3 As String = "")
            Kopfzeile = kopfzeile__1
            Textzeile1 = textzeile1__2
            Textzeile2 = textzeile2__3
            Bild = ""
            Parameter = ""
            LongRunning = False
            SoundLoop = False
            Silent = False
            Sound = "ms-winsoundevent:Notification.Looping.Alarm"
        End Sub
```

Über diese Eigenschaft können Sie abfragen, ob Popup-Benachrichtigungen überhaupt angezeigt werden und wenn nein, warum:

```
        Public ReadOnly Property Setting() As NotificationSetting
            Get
                Return ToastNotificationManager.CreateToastNotifier().Setting
            End Get
        End Property
```

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 Function CreateXML() As String
```

Falls eine Grafik vorhanden ist, ändert sich der Templatename:

```
            Dim templatename As String = "toast"
            If Bild <> "" Then
                templatename += "ImageAnd"
            End If
```

Mit Hilfe geschachtelter *XElement*-Konstruktoren erzeugen wir zunächst ein XML-Dokument:

```
            Dim xe As 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:

```
Dim bind = xe.Element("visual").Element("binding")
```

Falls ein Parameter vorhanden ist, wird dieser als Attribut eingefügt:

```
If Parameter <> "" Then xe.Add(New XAttribute("launch", Parameter))
```

Gleiches gilt für die Laufzeit der Anzeige (7 oder 25 Sekunden):

```
If LongRunning Then xe.Add(New XAttribute("duration", "long"))
```

Die Tonwiedergabe wird über ein zusätzliches Element *audio* gesteuert:

```
If SoundLoop And Not Silent Then _
    xe.Add(New XElement("audio", New XAttribute("loop", "true"),
            New XAttribute("src", Sound)))
```

Falls Ruhe erwünscht ist, geht auch dies:

```
If Silent Then 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 = "" Then
    ' nur body1
    bind.Attribute("template").Value = templatename & Convert.ToString("Text01")
    bind.Add(New XElement("text", New XAttribute("id", "1"), Textzeile1))
Else
```

Hier folgen die alternativen Varianten (siehe Kommentare):

```
    If Textzeile2 = "" Then
        ' nur einzeiliger Bodytext
        If Kopfzeile.Length > 20 Then
            bind.Attribute("template").Value = templatename & _
                Convert.ToString("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 & _
                 Convert.ToString("Text02")
            bind.Add(New XElement("text", New XAttribute("id", "1"), Kopfzeile))
            bind.Add(New XElement("text", New XAttribute("id", "2"), Textzeile1))
        End If
```

```
            Else
                ' zweizeiliger Bodytext
                bind.Attribute("template").Value = templatename & Convert.ToString("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))
            End If
        End If
```

Falls ein Bild definiert wurde, müssen wir dieses auch per Element einfügen:

```
        If Bild IsNot Nothing Then _
            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()
    End Function
```

Eine *ToastNotification* erzeugen:

```
    Public Function CreateToast() As ToastNotification
        Dim xdoc As New XmlDocument()
        xdoc.LoadXml(CreateXML())
        Return New ToastNotification(xdoc)
    End Function
```

Eine *ScheduledToastNotification* erzeugen (wir benötigen eine Zeitangabe):

```
    Public Function CreateScheduledToast(zeit As DateTimeOffset) As _
                        ScheduledToastNotification
        Dim xdoc As New XmlDocument()
        xdoc.LoadXml(CreateXML())
        Return New ScheduledToastNotification(xdoc, zeit)
    End Function
```

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 Function Show() As ToastNotification
        _toast = CreateToast()
        ToastNotificationManager.CreateToastNotifier().Show(_toast)
        Return _toast
    End Function
```

```
        Public Sub Hide()
            If _toast IsNot Nothing Then _
                ToastNotificationManag.CreateToastNotifier().Hide(_toast)
        End Sub

        Public Function AddToSchedule(zeit As DateTimeOffset) As ScheduledToastNotification
            Dim res As ScheduledToastNotification = CreateScheduledToast(zeit)
            ToastNotificationManager.CreateToastNotifier().AddToSchedule(res)
            Return res
        End Function

End Class
```

Test

HINWEIS: Siehe dazu Abschnitt 22.3.3 ab Seite 1120.

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	schnelle Distributionsmöglichkeit per Web
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
WinRT	Windows Runtime	Neues Framework seit Windows 8
WMI	Windows Management Instrumentation	Klassen zur Windows-Administration
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.vb	Quellcode für Webform
.cd	vom Klassen Designer angelegte Datei
.config	Konfigurationsdatei der Anwendung
.vbproj	VB-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
.vb	VB-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

.NET WinRT-Profil 924
.NET-Framework 66

A

Abbruchbedingung 624
Abort 519
Abs 308
Abstrakte Klassen 197
Abstrakte Methoden 198
Access Control Entries 444
Accessor 202
ACE 444
ACL 444
Activator 836
Add 680
AddAccessRule 444
AddAfterSelf 680
AddBeforeSelf 680
AddDays 304, 311
AddExtension 481
AddFirst 680
AddHandler 147
AddHours 304, 311
AddMinutes 304, 311
AddMonths 304, 311
AddRange 351
AddressList 876
AddressOf 147
AddressWidth 883
AddToSchedule 1164
AddYears 304, 311
Administrator 874
ADO.NET-Klassen 731
ADO.NET-Objektmodell 728
Aggregation 224
Aktionsabfrage 755
Aktivierung 1049
Alias 343
Alternator 291
Ancestors 677
And 116
Anfangswerte 95
Anonyme Methoden 362
Anonyme Typen 388
Anwendungsdienste 945
Anwendungseinstellungen 762
App.config 632, 762
App.xaml 1040
AppBar 1033, 1133
AppData 1059
Append 321, 465
AppendChild 686, 691, 692
AppendLinesAsync 1091

AppendText 467
AppendTextAsync 1091
Application 877
ApplicationDataContainer 1064
Arithmetische Operatoren 112
Array 145, 269, 331, 420, 425
ArrayList 350, 356
As 357
Assemblierung 57, 70
Assemblies 883
Assembly 70, 246
  dynamisch laden 836
  GetExecutingAssembly 833
  Laden 833
  LoadFrom 833
AssemblyInfo.vb 903, 1038
Assemblyinformation 1039
Assets 1047
Async 552
Asynchrone Programmierentwurfsmuster 542
Atn 308
Attribute 72, 653, 656, 684, 1162
Attributes 441, 1082
Auflistung 347
Ausgabefenster 620
Ausnahmen 642
Ausschneiden 829
Aussteuerungsbalken 889
Auswahlabfrage 757
Auto-Play 1069
Auto-Property 169
AutoPlay 1010
AutoResetEvent 611, 613
AvailablePhysicalMemory 883
AvailableVirtualMemory 883
Await 552

B

BackgroundWorker 539
Barrier 608
BatteryChargeStatus 882
BatteryFullLifetime 882
BatteryLifePercent 882
BatteryLifeRemaining 882
Beep 909
Befehlsdesign 957
Befehlsfenster 618
Befehlsleisten 957
BeginInvoke 535, 548
BeginUpdate 456
Begrüßungsbildschirm 1043
BEISPIEL 12.4: Prolog 652
BEISPIEL 12.5: DTD 652
Benannte Parameter 126
Benutzerschnittstelle 944
Bezeichner 88
BigInteger 371
Bild einlesen 851
Bilder drucken 853
Bildschirme 879
Binärdatei 470
BinaryFormatter 258, 464, 473
BinaryReader 464, 470, 471
BinarySearch 278
BinaryWriter 464, 470, 471
BindingNavigator 483
BindingSource 258, 483
BlockingCollection 608
Boolean 92, 103
BootMode 878
Border 986
BottomAppBar 1133
Boxing 110
Breakpoints 622
Button 976

ByRef 127
Byte 92, 100
Byte-Array 333, 718
ByVal 127

C

C# 141
Callback 544
Callback-Timer 559, 563
Caller Information 634
CallerFilePath 634
CallerLineNumber 634
CallerMemberName 634
CameraDeviceType 855
CancellationPending 540
CancellationToken 599, 610
CancellationTokenSource 600
Canvas 987
CaptureElement 1008
Case 117
Catch 144, 327
ChangeClipboardChain 843
ChangeDatabase 737
Char 92, 101
Char-Array 283
CharacterFormat.BackgroundColor 981
CharacterFormat.Size 981
CharmBar, jetzt per Win+H erreichbar) 1109
Chars 281
CheckAccess 536
CheckBox 978
CheckFileExists 450, 480
CheckPathExits 450
ChildNodes 689
Class 146, 154, 159
ClassesRoot 831
ClassLoader 69
Clear 278, 349
ClickOnce-Deployment 800
Clipboard 827
Clone 277, 287, 768
Close 736
ClosedByUser 1050
Cloud 1066
CLR 67, 69
CLR-Threadpool 582
CLR-Version 885
CLS 67
Code Contracts 646
Code Document Object Model 244
Code Manager 69
CodeDOM 244
CodeDomProviders 244
Codefenster 64
Collection 347, 355, 426
Collection-Initialisierer 271
CollectionBase 699
CollectionViewSource 1022, 1030
ColumnName 771
Columns 769, 771
ColumnSpan 990
COM-Komponenten 71
COM-Marshaller 70
ComboBox 998
Command 738, 750
CommandBuilder 745
CommandLine 884
CommandText 739
CommandTimeout 739
CommandType 740
Common Language Runtime 64, 69
Common Language Specification 67
Common Type System 67, 68
CompanyName 884
CompareDocumentOrder 677
CompiledAssembly 247
Complex 373

CompositeTransform 983
ComputerInfo 878
ComputerName 881
ConcurrentBag 608
ConcurrentDictionary 608
ConcurrentQueue 608
ConcurrentStack 608
Connection 732, 739
ConnectionString 734, 762
ConnectionStringBuilder 737
ConnectionTimeout 736
Console
  BackgroundColor 915
  Clear 916
  Title 916
Console.In 913
Console.Out 913
ConsoleKeyInfo 912
Const 97
Constraint 357
ContactPicker 1114
Contains 349
Contract 922, 1038
Convert 108, 137
ConvertStringToByteArray 496
ConvertXMLToDataSet 719
Copy 440, 767
CopyAsync 1094
Copyright 884
CopyTo 277, 287, 349, 441
Cos 308
Count 403
CountdownEvent 608
Create 466
CreateCommand 737, 739
CreateDirectory 437
CreateElement 687
CreateEncryptor 475
CreateFileAsync 1089
CreateFolderQueryWithOptions 1084
CreateFromFile 478
CreateGraphics 892
CreateInstance 247
CreateNavigator 702, 723
CreateNew 478
CreateSubdirectory 437
CreateSubKey 832
CreateText 466
CreateToastNotifier 1161
CreateViewAccessor 479
CreationCollisionOption 1061, 1086
CreationTime 441
Credentials 1068
CRUD 780
CryptographicException 497
CryptoStream 475
CryptoStreams 496
CSV-Datei 506, 509
CTS 67
CType 107
CultureInvariant 292
Currency 92
CurrentClockSpeed 883
CurrentConfig 831
CurrentCulture 878
CurrentDirectory 884
CurrentStateChanged 1010
CurrentUser 831
CursorLeft 909
CursorTop 909
CursorVisible 909
Custom Events 175

D

Data Encryption Standard 496
DataAdapter 749
Database 735

DataChanged 1067
DataColumn 769
DataContext 794
DataGrid 718
DataGridView 258
DataPackage 1103, 1109
DataPackageView 1103
DataReader 746
DataRequested 1109
DataSet 718
DataSource 735
DataTable 769
DataTemplate 1006
DataTime 303
DataTransfer 1103
DataTransferManager 1109, 1110
DataView 773, 781
Date 92, 102
DateCreated 1082
Datei komprimieren 499
Datei verschlüsseln 496
Dateiattribute 441
Dateien umbenennen 441
Dateiname 511
Dateiparameter 465
Dateipicker 1095
Dateitypzuordnung 1075
Dateiverknüpfung 1075
Dateiverknüpfungen 840
Datenkonsument 727
Datenprovider 727, 728
Datenquelle 776, 795
Datentypen 92, 100, 143
Datentypzuordnung 1070
Datenzugriff 122
DateTime 92, 303
Datumsformatierung 313
Day 303, 310
DayOfWeek 303, 310
DayOfYear 303, 310
DaysInMonth 305, 312
DbProviderFactories 730
Deadlocks 515
Debug 627
  Write 628
  WriteIf 628
  WriteLineIf 628
Debuggen 1054
Debugger 617
DebugView 1056
Decimal 101
Declare 146
Decrypt 474
DefaultExt 480
DefaultFileExtension 1097
Deklarationen 1045
Delegate 359, 379
Delegate instanziieren 361
Delete 437, 772
DeleteAsync 1086, 1094
DeleteCommand 750
DeleteContainer 1068
DeleteSubKey 832
DeleteSubKeyTree 832
DeleteValue 832
Delta 984
Deployment 801
DereferenceLinks 450
DES 496
Descendants 677
Description 884
DESCryptoServiceProvider 475, 496
Designer 63
DesktopDirectory 451
Destruktor 178, 181
DeviceManager 848
DialogResult 481
Dictionary 356

DictionaryEntry 881
Dim 94
Direction 745
Directory 432, 436, 437
DirectoryInfo 432, 437
DirectoryName 436
DirectorySecurity 444
DisplayName 1082
DisplayType 1082
Dispose 183, 743
Distinct 418
Do 119
Do Until...Loop 134
Do While 119, 144
Do While...Loop 134
Do...Loop 119
Do...Loop Until 134
Do...Loop While 134
DOCTYPE 656
Document Object Model 684
Document Type 684
Document Type Definition 652
DocumentsLibrary 1063
DOM 684
Double 92, 101
DownloadsFolder 1062
DragItemsStarting 1003
DriveInfo 432
DynamicObject 367
dynamische Eigenschaft 382
dynamische Programmierung 366
dynamisches Objekt 382
DynData 831

E

Eigenschaften 163
Eigenschaften-Fenster 63
Eigenschaftsmethoden 230
Einfügen 829
Einzelschritt-Modus 625
Element 653, 656, 684
Elements 677
Else 117
ElseIf 116, 117, 144
EnableRaisingEvents 447
Encoding 333
Encrypt 474
EndInvoke 535, 548
EndsWith 281
Energiestatus 882
Enter 527
Entwicklungsumgebung 61
Enum 145
Enumerable 420
Enumerationen 145
Environment 878, 885, 907
Environment Variablen 881
Equals 419
Erase 276
Ereignis 147, 172
Ereignisse 59
Erweiterungsmethoden 390, 393, 408
Escape-Zeichen 293
Event 147, 172
EventLog 633
EventLogTraceListener 633
Events 59
Exception 325, 643
ExceptionManager 69
ExecutablePath 884
ExecuteNonQuery 738, 741
ExecuteReader 738, 741, 747
ExecuteScalar 738, 742
Exists 441
Exit 527
Exit Do 134
Exp 308

ExpandoObject 368
Exponentialfunktion 309
ExtClock 883
Extension 436, 1038
Extension Method-Syntax 392, 408

F

FailIfExists 1062
Families 879
Fast and Fluid 921
Fehler 325
Fehlerbehandlung 635
Fehlerklassen 636, 644
Fehlerstatus 905
Fehlersuche 85
FieldCount 748
File 432, 464
File Type Association 1075
FileAccess 465
FileAttribute 442
FileDropList 827
FileExtension 852
FileInfo 432, 464, 467
FileIO 1090, 1092
FileMode 465
FileName 450, 481
FileOpenPicker 1095
FileSavePicker 1095, 1097
FileSecurity 444
FileShare 466
FileStream 464
FileSystemAccessRule 444
FileSystemWatcher 432, 447
FileTypeChoices 1097
Fill 751
Filter 447, 480, 481
FilterIndex 450
Filtern 773
Filters 450
Finalize 182
Find 772
Fingereingabe 960
FirstChild 688, 711, 722
FlipView 1006
Flyout 1127
FolderBrowserDialog 451
FolderPicker 1095, 1098
FolderRelativeId 1082
FontFamily 879
Fonts 451
FontSmoothingContrast 879
FontSmoothingType 879
For 119, 144
For Each 145, 271, 356
for-each 710
For...Next 119, 133
Form1.vb 63
Format 314
Formelparser 331
Formular 58
Frame 955, 1011
Freigabeziel 1112, 1154
Friend 153
From 396
FromCurrentSynchronizationContext 604, 615
FullName 436
Function 124, 145
Funktionen 124, 145, 1044
Funktionsdiagramm 249
FutureAccessList 1099

G

Garbage Collector 181
GenerateUniqueName 1061, 1086
Generics 352, 354
generische Schnittstelle 419

Geräteeigenschaften 850
Gestensteuerung 983
Get 163, 202
GetBitmapAsync 1103
GetChanges 768
GetCommandLineArgs 906
GetCreationTime 441
GetCurrent 875
GetCurrentDirectory 438
GetDataAsync 1103
GetDataObject 827
GetDataPresent 828
GetDefaultView 1030
GetDirectories 438
GetDrives 439
GetDynamicMemberNames 367
GetElementsByTagName 693
GetEnumerator 349
GetEnvironmentVariables 881, 907
GetExecutingAssembly 884, 1040
GetFactoryClasses 730
GetFields 834
GetFiles 443
GetFoldersAsync 1084
GetHostEntry 876
GetHostName 875
GetLength 277, 287
GetMembers 834
GetMethods 835
GetProcessById 572
GetProperties 835
GetStorageItemsAsync 1103, 1157
GetSubKeyNames 832
GetTextAsync 1157
GetValue 748
GetValueNames 832
GetValues 748
Global 343
Glühbirne 85
GPS 944
Grafikbearbeitung 856
Grid 989
GridView 1004
GridView gruppieren 1020
GroupBy 400
Gruppen 871
GZipStream 499

H

Haltepunkte 623
Hardware-Informationen 882
Hashtable 224, 351
Hauptprogramm 904
HeaderTemplate 1022
Help 822
HelpMaker 825
HelpNameSpace 824
HelpProvider 824
Hilfe-IDE 825
Hilfedatei 817, 821
Hilfemenü 822
HomeGroup 1063
Hour 303, 310
Hover 976
HTML 649
HTML 5 954
HTML Help Workshop 818
HtmlFormatHelper 1104
HyperlinkButton 976

I

IAsyncResult 544
ICollection 349
IComparable 253
IComparer 253
ICryptoTransform 497

IDataObjekt 828
IDisposable 184, 743
Idle-Prozesse 571
IEnumerable 348, 354, 420
IEqualityComparer 419
If 116, 144
IgnorePatternWhitespace 292
IgnoreWhitespace 701
IInspectable 946
IList 349
ILSpy 925
Image 983
ImageFile 852, 856
ImageProcess 856
Implizite Zeilenfortsetzung 90
ImportRow 770
Imports 156, 341
Indent 708
IndentChars 708
IndentLevel 629
IndentSize 629
Index 269
Indexer 227, 230, 345
IndexOf 278, 281, 317, 349
Inherits 188
InitialDirectory 481
Initialisierer 420
Initialisierung 157
Initialize 277, 287
InnerText 694
InputScopeName 1018
Insert 281, 349
InsertCommand 750
Installationsverzeichnis 1061
InstallShield 808
Instanziieren 156
Int16 92
Int32 92
Int64 92
Integer 92, 100
Intellisense 160
Interface 203
InteropServices 895
Interrupt 519
InvalidOperationException 326
Invoke 534, 546, 548, 836
InvokeRequired 536
IP-Adresse 875
IsAbstract 834
IsActive 984
IsAdmin 874
IsAfter 678
IsAlive 520
IsBackGround 520
IsBefore 678
IsClass 834
IsClosed 748
IsCOMObject 834
IsCompleted 588
IsEnum 834
IsFixedSize 349
IsFontSmoothingEnabled 879
IsInterface 834
IsLeapYear 305, 312
IsLooping 1010
IsMatch 289
IsMuted 1010
IsPublic 834
IsReadOnly 349
IsSealed 834
IsSourceGrouped 1021
IsSynchronized 349
Item 349, 748
ItemsControl 998
ItemsPanelTemplate 1022
Iterationsschleife 138
Iterator 588
iTextSharp 503

IUnknown 946

J

JavaScript 954
JIT-Compiler 65
Join 284, 331, 402, 519

K

Kapselung 152
Kartenspiel 264
Kartesische Koordinaten 236
Klasse 151
Klassendefinition 146
KnownFolders 1063
Kommandozeilenparameter 906
Kommentare 89, 142, 653
komplexe Zahlen 236
Komponenten 59
Komposition 224
Komprimieren 476
Konfigurationsdatei 762
Konsolenanwendung 53, 903
Konstanten 92, 97
Konstruktor 178, 571
Konstruktor überladen 230
Kontaktliste 1113
Kontextmenü 840
Kontravarianz 370
Konvertieren 106
Konvertierungsoperatoren 346
Kopieren 829
Kovarianz 370
kritische Abschnitte 559

L

Lambda Expression 379
Lambda-Ausdrücke 363, 393, 408
Language Projection 924, 946
LastAccessTime 441
LastWriteTime 441
Late Binding 370
Laufwerke 439
Launch-Contract 1049
Layout-Control 987
Lebenszyklus 1047
Length 277, 281, 286, 287
LINQ 387, 425, 426, 428, 509
  Abfrage-Operatoren 394
  Aggregat-Operatoren 402
  AsEnumerable 405
  Count 402
  GroupBy 400
  Gruppierungsoperator 400
  Join 401
  Konvertierungsmethoden 405
  OrderBy 398
  OrderByDescending 398
  Projektionsoperatoren 396
  Restriktionsoperator 398
  Reverse 400
  Select 396
  SelectMany 396
  Sortierungsoperatoren 398
  Sum 403
  ThenBy 398
  ToArray 405
  ToDictionary 405
  ToList 405
  ToLookup 405
  Verzögerte Ausführung 404
  Where 398
LINQ to Entities 780
LINQ to SQL 780
LINQ to XML-API 672
LINQ-Abfrageoperatoren 391

LINQ-Provider 388
LINQ-Syntax 391
List 355
ListBox 332, 998
ListView 1002
Load 675
LoadCompleted 1012
LoadedAssemblies 884
LoadXml 685
LocalApplicationData 451
LocalFolder 1059
LocalMachine 831
LocalState 1059
Log 308
Log10 308
Logarithmus 309
LogicalDpi 967
Logische Operatoren 114
Lokal-Fenster 619
Long 92, 100
LongRunning 603
Lookahead 298
Lookbehind 299
Loop 144
Loop While 144
LowestBreakIteration 588

M

MachineName 881
ManagementObjectSearcher 871
ManipulationDelta 983
ManipulationMode 983
ManualResetEvent 613
ManualResetEventSlim 608
Manufacturer 883
Map View 478
Mapperklassen 794
Match 289, 290
Matches 289, 291
Matrix 230
Matrizen 235
Max 308
MaximumRowsOrColumns 990
MCI 886, 895
MediaCapture 1008
MediaElement 1009
MediaEnded 1010
MediaOpened 1010
Mehrfachdeklaration 94
Memory Mapped File 477, 490
MemoryMappedFile 478
MemoryStream 718, 789
MenuStrip 877
MessageBox 129
MessageDialog 1035, 1115
Messwertliste 411
Metadaten 71, 925
Metasprache 649
Metazeichen 293
Method-Overriding 186
Methoden 59, 124, 145, 169
  generische 358
  überladen 136, 230
Methodenzeiger 359
MethodImpl 531
MethodInfo 246
Methods 59
Microsoft Intermediate Language Code 65
Mikrofon 888
Mikrofonpegel 889
Min 308, 490
Minute 303, 310
MMF 478
Mod 115
Module 57
Monitor 527
MonitorCount 880

MonitorsSameDisplayFormat 880
Month 303, 310
MostRecentlyUsedList 1101
Move 438, 440
MoveAndReplaceAsync 1094
MoveAsync 1094
MoveBufferArea 909
MoveTo 441
MoveToNext 702, 722
MoveToPrevious 702, 722
MoveToRoot 722
MRU-Liste 1102
MSIL-Code 65
Multiselect 450
Multitasking 514
Multithreading 73, 514, 558
MusicLibrary 1063
MustInherit 197
MustOverride 198
Mutex 530
My Project 63
MyBase 187
MyComputer 451
MyDocuments 451
MySettings 764

N

Namespace 70, 156, 343, 834
Narrowing 346
Navigate 961
NavigateToString 1011
Navigation 961
NavigationCacheMode 962
Navigationsdesign 959
NET-Reflection 833
Network 881
Netzwerk 881
New 146, 178
NewRow 770
Next 422
NextMatch 290
NextSibling 689, 711
Nodes 678
NodeType 706
Not 116
Nothing 158
Notification 944
NotifyFilter 447
NotInheritable 198
NotRunning 1050
Now 305, 312
NTFS 475
Nullable Type 111
Nullbedingter Operator 202
Nutzer 871

O

Object 92, 103
Objekt 151
Objekt-Initialisierer 157, 180
Objektbaum 483
Objektbäume 258
Objekte 147
Objektinitialisierer 209
Objektrelationales Mapping 780
OleDbConnection 732
OnActivated 1050
OnCachedFileUpdaterActivated 1050
OnFileActivated 1050, 1070, 1071
OnFileOpenPickerActivated 1051
OnFileSavePickerActivated 1051
OnLaunched 961, 1040, 1049
OnNavigatedTo 962, 1157
OnSeachActivated 1051
OnShareTargetActivated 1051, 1113, 1156
OnSuspending 1040

OOP 73, 149, 264
OOWLib 864
Open 466, 736
OpenFileDialog 449, 480
OpenOffice.org 863
OpenOrCreate 466
OpenReadAsync 1158
OpenText 468
Operatoren 89, 112, 143
Operatorenüberladung 236, 344
Option Explicit 95
Option Strict 96
Optional 128
Optionale Parameter 128
OrderBy 398
ORM 780
OSFullName 878
OSVersion 878
Overrides 187, 189
OverwritePrompt 481

P

Package.appxmanifest 1041
PadLeft 281
PadRight 281
Page 955
Paragraph 976
Parallel LINQ 364, 608
Parallel-Programmierung 579
Parallel.For 585
Parallel.ForEach 589
Parallel.Invoke 583
ParallelLoopResult 588
Parameter 744
ParameterName 744
Parameterübergabe 126, 906, 961, 968
ParentNode 711
Parse 108, 305, 312, 676
Parser 656
Partielle Klassen 199
PasswordBox 979
PasswordVault 1068
Path 432, 446
Pattern 294
Pattern Matching 288
Pause 1010
PC-Name 881
PDF 502
PDFsharp 504
PeekChar 470
Pegeldiagramm 891
PerformanceData 832
PI 308, 656
PickMultipleContactsAsync 1114
PickSaveFileAsync 1097
PicturesLibrary 1063
PlacementTarget 1012
Platform 878
Play 1010
PLINQ 405, 608
PointToScreen 823
Polarkoordinaten 236
Polling 543
Polymorphes Verhalten 195
Polymorphie 152, 184, 197
PopUp 1127
Popup-Benachrichtigungen 1120
PopUp-Hilfe 823
PopupMenu 1131
portieren 141
Position 1010
Potenz 309
Pow 308
PowerLineStatus 882
PowerStatus 882
PreferFairness 603
Preserve 273

Press 976
PreviousExecutionState 1050
PreviousSibling 711
PrimaryMonitorMaximizedWindowSize 880
PrimaryMonitorSize 880
Priority 520
Private 99, 153
Procedure-Step 622
Process 570, 575
Process Sandboxing 922
Process.Start 576
Processing Instruction 652, 656, 684
ProcessorCount 883
ProcessThread 570
ProductName 884
ProductVersion 884
Program.vb 903
Programm starten 575
Programmablaufplan 139
ProgressBar 889, 984
ProgressChanged 540
ProgressRing 984
Projektmappen-Explorer 62
Projekttyp 62
Properties 59
Property 146, 163, 206
Protected 153
Protected Friend 153
Protokolldatei 1054
Provider 732, 735
Prozedur 125
Prozeduren 145
Prozedurschritt 626
Prozesse 570
Public 100, 153
Public Property 168
Pulse 527, 528
PulseAll 527, 528

Q

Quantifizierer 289, 295
Query Expression-Syntax 392, 408
QueryOptions 1084
Queue 354, 356
QueueUserWorkItem 522

R

Racing 515
RadioButton 978
RaiseEvent 147, 173
Random 422
RandomAccessStreamReference 1105
Rank 277, 286, 287
Read 465
ReadAllBytes 470
ReadAllLines 468
ReadAllText 468
ReadBufferAsync 1091
ReadContentAsFloat 707
ReadKey 909, 912
ReadLine 57
ReadLines 468
ReadLinesAsync 1091
ReadOnly 166
ReadOnly-Eigenschaft 169
ReadTextAsync 1091
ReadToEnd 468
ReadWrite 465
ReadXml 696, 784, 789
Recordvariable 123
ReDim 273
Referenzieren 156
Referenztypen 280
Reflection-Mechanismus 245
Reflexion 71
Regedit.exe 830

Regex 289
RegexOptions 292
Registrierungsdatenbank 830
Registrierungseditor 830
Registry 829, 831, 838, 878
RegistryKey 829, 831
Reguläre Ausdrücke 288
RegularExpressions 288
Relationen 718
Relaxed Delegates 362
Release 976
ReleaseMutex 530
Remote-Debugging 931
RemovableDevices 1063
Remove 281, 349, 681, 772
RemoveAccessRule 444
RemoveAll 681
RemoveAnnotations 681
RemoveAt 349
RemoveAttributes 681
RemoveContent 681
RenderTransform 983
Repeat 422
RepeatButton 976
Replace 281, 289
ReportStarted 1157
ResolutionScale 967
Resume 519
Resuming 1052
RetrievePropertiesAsync 1083
RichEditBox 979
RichTextBlock 973
RightToLeft 292
Roaming 1066
Roamingdaten 1068
RoamingFolder 1060
RoamingStorageQuota 1060
Rotation 984
Round 308
RowFilter 774
Rows 771
RowSpan 990
Rückrufmethode 543
Rücksprung 626
Run 976
Running 1047

S

Sandboxing 920
SaveFileDialog 449, 480
Scanner 859
Scanner-Assistent 854
ScannerDeviceType 855
Schaltjahr 305
ScheduledToastNotification 1163
Schleifen 144
Schleifenabbruch 587
Schleifenanweisungen 119, 133
Schlüsselwörter 88
Schnittstelle 203
Schriftarten 879, 960
ScreenOrientation 880
ScrollBar 984
ScrollViewer 988
Second 303, 310
Security Engine 69
Seitennavigation 968
Seitenstatus 962
SELECT 396, 702, 757
Select Case 117, 144
SelectCommand 750
SelectMany 397
SelectNodes 693
SelectSingleNode 689, 692, 693, 722
SemanticZoom 1025
Semaphore 532
SemaphoreSlim 608

SendMessage 843
Sequenzielle Datei 471
Serialisieren 472
Serialisierung 72
Serializable 258, 472
ServerVersion 735
ServicePack 878
Set 163, 202
SetAccessControl 445
SetAttributeValue 680
SetBitmap 1103, 1105
SetBufferSize 909
SetClipboardViewer 843
SetCurrentDirectory 438
SetCursorPosition 909
SetData 1103
SetDataObject 827
SetElementValue 680
SetSource 1010
SetStorageItems 1103, 1106
SetText 981
Settingsbereich 959
Setup-Projekt 808
SetWindowPosition 909
SetWindowSize 909
Shared 146, 167, 171
Shared-Methoden 230
ShareOperation 1156
Short 100
ShowAcquireImage 863
ShowAcquisitionWizard 855
ShowDialog 481
ShowForSelectionAsync 1131
ShowHelp 822
ShowPopup 823
ShowSelectDevice 860
ShowShareUI 1111
Sichtbarkeit 153
Sign 308
Sin 308
Single 92, 101
Single-Step 622
SizeChanged 964
Skip 679
SkipWhile 679
Sleep 519
Slider 984
SnapsTo 985
SocketDesignation 883
Sort 253, 278, 773
SortedList 356
SortedSet 374
Sortieren 425, 773
Sound 886, 895
Soundkarte 884
SpecialFolder 451
Sperrmechanismen 523
SpinLock 608
SpinWait 608
Splash Screen 1043, 1051, 1073
Split 281, 284, 289, 331
SQLite 1137
Sqr 308
Stack 354
StackPanel 988
Stammelement 654
StandardDataFormats 1157
Standardeigenschaft 225
StartPreviewAsync 1009
Startseite 961
StartsWidth 281
State 736
statische Klasse 209
Statische Methoden 171
Statusmitteilung 1122
StepValue 985
Stop 1010
Stopped 1047

Storage-Interface 944
StorageApplicationPermissions 1099
StorageDeleteOption 1087
StorageFile 1089, 1093
StorageFileQueryResult 1152
StorageFolder 1081
Store 926
StoredProcedure 740
Streamdaten 913
StreamReader 464, 468, 1092
StreamWriter 464, 467, 1091
String 92, 101, 331, 333
Stringaddition 319
StringBuilder 285, 319
StringFormatConverter 1015
Stringinterpolation 316
StringReader 464
Stringvergleich 284
StringWriter 464
Structure 121, 145, 336
Strukturen 145
Strukturvariablen 335
Sub 125, 146
Sub Main 904
SubKeyCount 832
Subklassen 187
SubmitChanges 797
Substring 281, 317
Success 290
Suchen 774
Suchfunktionen 331
SuggestedStartLocation 1099
Sum 403
Suspend 518
Suspended 1050
Suspending 1047, 1051
SuspendingDeferral 1052
SVG 983
SyncLock 524
SyncRoot 349
System 92, 451
System.Console 908
System.Dynamic 367
System.Environment 877
System.IO.Compression 476
System.IO.FileStream 463
System.IO.Stream 463
System.Management 871
System.Net 875
System.Object 201
System.Reflection 833
System.Security.AccessControl 444
System.Security.Cryptography 475
System.Security.Principal 874
System.Text 334
System.Threading 517, 582
System.Threading.Tasks 582
System.XML 685
System.Xml.Linq 672
System.Xml.Serialization 701
SystemInformation 877

T

TableAdapter 777
TableDirect 740
TableName 771
Tablet-Simulator 984
Take 679
TakeWhile 679
Tan 308
Task
  Canceled 603
  ContinueWith 596, 604
  Created 603
  Datenübergabe 592
  Faulted 603
  Fehlerbehandlung 602

IsCanceled 603
IsCompleted 603
IsFaulted 603
RanToCompletion 603
Result 596
Return 598
Rückgabewerte 595
Running 603
Status 603
Task-Ende 604
Task-Id 602
TaskCreationOptions 603
Taskende 593
Userinterface 604
Verarbeitung abbrechen 598
Wait 594
WaitAll 595
WaitingForActivation 603
WaitingForChildrenToComplete 603
WaitingToRun 603
Weitere Eigenschaften 602
Task Parallel Library 364
Task starten 591
Task.Factory.StartNew 590
TaskCreationOptions 610
TaskScheduler 604, 610
Tastaturabfrage 912
Tasteneingaben 1019
TemporaryFolder 1061
TemporaryKey.pfx 1046
TerminalServerSession 881
Terminated 1050
TextBlock 973
TextBox 979
Textdatei 382, 466, 479
TextWriterTraceListener 631
Then 116, 144
ThenBy 399
ThenByDescending 399
Thin Client 211
Thread 517, 518, 558
initialisieren 558
synchronisieren 559
Thread Service 70
Thread(Of T) 556
ThreadInteruptedException 519
ThreadPool 521
Threads 570
threadsicher 533
ThreadState 520
ThreadWaitReason 573
Throw 325, 326, 638, 643
ThrowIfCancellationRequested 600
Thumbnail-Ansicht 1096
TickValues 985
Tiles 920
Timer 889
Timer-Threads 538
TimeSpan 319
Title 450, 481, 884
ToArray 417
ToastNotificationFactory 1121
ToastNotificationManager 1161
ToastNotifications 1159
ToCharArray 281
Today 305, 312
ToDouble 137
ToggleButton 978
ToggleSwitch 978
ToLongDateString 304, 311
ToLongTimeString 304, 311
ToLower 281
ToolTip 1012
ToolTipPlacement 1012
ToolTipService 1012
TopAppBar 1133
ToShortDateString 304, 311
ToShortTimeString 304, 311

ToString 107, 313, 321
TotalPhysicalMemory 883
TotalVirtualMemory 883
Touchscreen 939
ToUpper 281
TPL 582
TPL Datenstrukturen 607
Trace 627, 630
TraceListener 631
Trademark 884
Transform 709
Transformationsdatei 709
TranslateX 984
TranslateY 984
TreeView 455, 714
Trefferanzahl 625
Trennzeichen 332
Trim 281
Truncate 466
Try 144
Try-Catch 325, 635
Try-Finally 640
TryBinaryOperation 367
TryCast 111
TryConvert 367
TryCreateInstance 367
TryEnter 527, 530
TryGetIndex 367
TryGetMember 367
TryInvokeMember 367
TrySetIndex 367
TrySetMember 367
Tuple 374
Type 834
Typecasting 379
TypeOf 205
Typinferenz 408
Typisierte DataSets 774
Typsicherheit 353

U

Überladen 128, 344
Überladene Methoden 170
Überwachungsfenster 620
Uhr anzeigen 305
UICommand 1131
Umgebungsvariablen 54, 907
UML 184
Unboxing 110
Unicode 101
UnicodeEncoding 496
Unified Modeling Language 184
UnIndent 629
UnspecifiedDeviceType 855
Unterverzeichnis 438
Update 746, 752
UpdateCommand 750
Updates 803
User-Name 881
UserDomainName 881
UserInteractive 881, 884
UserName 881
UserProfile 451
Users 832
Using 184, 469, 490, 491, 743
UTF-16 656
UTF-8 656

V

ValidateNames 481
Value 744
ValueCount 832
Variablen 92
Variablentypen 92
VariableSizedWrapGrid 990, 1023
VB-Compiler 53
VB-Source-Datei 55

vbc.exe 53
Verarbeitungsstatus 588
Vererbung 152, 224, 229
Vergleichsoperatoren 115
Veröffentlichen 804, 805
Verpacken 1046, 1145
Verschlüsseln 473
Versiegelte Klassen 198
VersionString 878
Vertrieb 1145
Verweistypen 93
Verzweigungen 144
Video 895
VideoDeviceType 855
VideosLibrary 1063
ViewManagement 964
ViewMode 1096
VirtualScreen 880
Virtuelle Tastatur 1017
Visual Studio 51
Visual Studio Enterprise 52
Visual Studio Professional 52
VisualStateManager 965
Vollbildmodus 926
Volume 1010

W

W3C 684
Wait 527, 528
WaitOne 530, 532, 612
WAV 886
Webcam 857, 944
Webpublishing-Assistent 801, 807
WebView 1011
Wend 119
Wertetypen 93
Where 398, 679
While 119
WIA 845
wiaaut.dll 846
Widening 346
Wiederholmuster 423
Wiederverwendbarkeit 152
Wildcards 294
Win32_SoundDevice 884
Window 955
Windows 10 928
Windows App 1037
Windows Management Instrumentations 877
Windows Runtime 919
Windows Store 926
Windows Store App 920
Windows-Philosophie 57
Windows-Simulator 928
WindowsIdentity 872, 875
Winkel 309
WinMD 948
WinMD-Dateien 924
winmm.dll 896
WinRT 919, 941
WinRT-API 943
WinRT-COM 924
WinRT-Namespaces 945
With 123
WithEvents 147, 173
WMI 877
work stealing 583
WorkerReportsProgress 540
WorkingArea 880
WorkingSet 884
WPF 536
Write 465
WriteAllBytes 470
WriteAttributeString 707
WriteBufferAsync 1091
WriteEndDocument 708
WriteEndElement 707

WriteIf 627
WriteLine 57, 627
WriteOnly 166
Writer 863
WriteStartDocument 707
WriteTextAsync 1091
WriteXml 784, 789
WriteXmlSchema 696
Wurzel 309

X

XAML 954
XAttribute 673, 1163
XComment 673
XDeclaration 673
XDocument 672, 675
XDocumentType 673
XElement 673, 1163
XLINQ 667
XML 649
XML transformieren 681
XML-Schema 662
XML-Strings 718
XmlAttribute 685, 701
XmlCDATASection 685
XmlCharacterData 685
XmlComment 685
XmlDataDocument 696
XmlDocument 696, 702, 711, 723, 1163
XmlDocumentType 685
XmlElement 685, 701
XmlEntity 685
XmlEnum 701
XmlIgnore 701
XmlImplementation 685
XmlNamedNodeMap 684
XmlNode 684, 688
XmlNodeList 684
XmlParseError 685
XmlProcessingInstruction 685
XmlReader 705, 717
XmlRoot 701
XmlSerializer 699
XmlText 685
XmlWriter 707
XmlWriterSettings 708
XNode 673
Xor 116
XPathDocument 702
XPathNavigator 702, 722, 723
XPathNodeIterator 702
XProcessingInstruction 673
XSD-Schema 660
xsd.exe 666
XslCompiledTransform 709
XSLT 709

Y

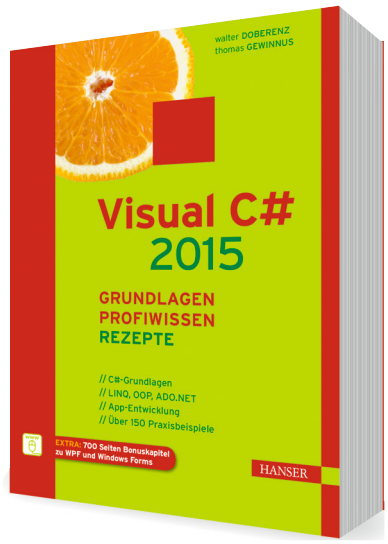
Year 303, 310
Yield 349

Z

Zahlenformatierung 313
Zeichenmengen 294
Zeilenumbruch 90
Zeitmessung 319
Zielplattformen 927
ZoomedInView 1026
ZoomedOutView 1026
Zufallszahlen 422
Zugriffsberechtigung 444
Zugriffsmodifizierer 153
Zuweisungsoperatoren 113
Zwischenablage 827, 842, 1102

HANSER

Der Klassiker zu C#



Doberenz, Gewinnus

Visual C# 2015

Grundlagen, Profiwissen und Rezepte

1195 Seiten

€ 49,99. ISBN 978-3-446-44381-5

Auch als E-Book erhältlich

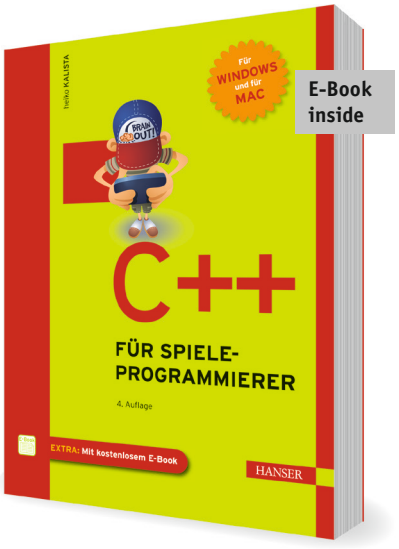
€ 39,99. E-Book-ISBN 978-3-446-44606-9

Dieser Klassiker der C#-/.NET-Programmierung bietet Ihnen Informationen und Lösungen, nach denen Sie anderswo vergeblich suchen. Einsteiger erhalten ein umfangreiches Praktikum zu den Grundlagen der .NET-Programmierung mit Visual C# 2015, dem Profi liefert es fortgeschrittene Programmiertechniken zu allen wesentlichen Einsatzgebieten der Windowsprogrammierung. Zum sofortigen Ausprobieren finden Sie, in Themengruppen übersichtlich angeordnet, über 150 qualitativ hochwertige Rezepte.

Mehr Informationen finden Sie unter **www.hanser-fachbuch.de**

HANSER

Let's code!



Kalista
C++ für Spieleprogrammierer
4., aktualisierte Auflage
479 Seiten. E-Book-inside
€ 34,99. ISBN 978-3-446-43216-1
Auch einzeln als E-Book erhältlich
€ 27,99. E-Book-ISBN 978-3-446-43395-3

- Für Einsteiger in die Spieleprogrammierung, die C++ von Grund auf erlernen oder ihre C++-Kenntnisse vertiefen wollen
- Mit vielen Beispielen ausschließlich aus der Spieleentwicklung
- Garantierter Lernerfolg durch: Aufgabenstellung – Tipps für die Lösung – Musterlösung
- Online: Leseforum und Codebeispiele aus dem Buch

Mehr Informationen finden Sie unter **www.hanser-fachbuch.de**

HANSER

Spiele entwickeln leicht gemacht

E-Book inside

Seifert
Spiele entwickeln mit Unity 5
2D- und 3D-Games mit Unity und C#
für Desktop, Web & Mobile
2., aktualisierte und erweiterte Auflage
631 Seiten. Komplett in Farbe. E-Book-inside
€ 39,99. ISBN 978-3-446-44563-5
Auch einzeln als E-Book erhältlich
€ 31,99. E-Book-ISBN 978-3-446-44580-2

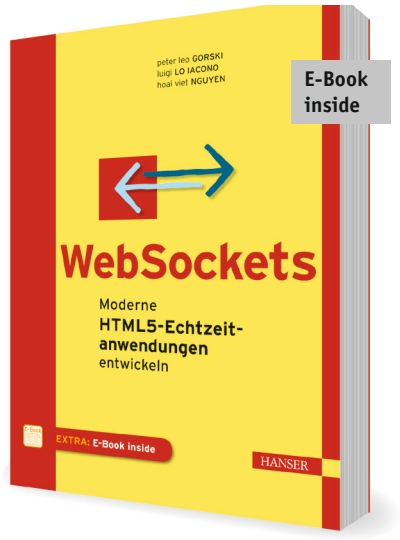
- Für alle, die ihr eigenes Spiel entwickeln wollen; Vorkenntnisse sind nicht erforderlich
- Im Internet: Die Games aus dem Buch mit allen Ressourcen, Videotutorials zur Vertiefung und weitere Beispiele, Zusatzmaterialien
- Extra: E-Book inside

Entwerfen Sie komplexe 3D-Welten; kombinieren Sie die unterschiedlichen Tools und Techniken, die Unity bietet; entwickeln Sie zusammen mit dem Autor zwei komplette Spiele.

Mehr Informationen finden Sie unter **www.hanser-fachbuch.de**

HANSER

Dynamisch, schnell, live!



Gorski, Lo Iacono, Nguyen
WebSockets
Moderne HTML5-Echtzeitanwendungen entwickeln
281 Seiten. E-Book-inside
€ 34,99. ISBN 978-3-446-44371-6

Auch einzeln als E-Book erhältlich
€ 27,99. E-Book-ISBN 978-3-446-44438-6

- Für Webentwickler mit Grundkenntnissen, Studenten der Informatik, Softwarearchitekten und IT-Konzepter
- Umfassender Einstieg in die WebSocket-Technik: Grundlagen, Performance, Sicherheit, Test-Verfahren
- Mit zahlreichen Beispielen auf Basis ganz unterschiedlicher Werkzeuge, Technologien, Sprachen und Frameworks
- Die Beispiele aus dem Buch sowie Infos zu aktuellen Entwicklungen können auf der Website der Autoren heruntergeladen werden

Mehr Informationen finden Sie unter **www.hanser-fachbuch.de**

VISUAL BASIC 2015 //

- **Für Einsteiger, Fortgeschrittene und Profis**
- **Übungs- und Nachschlagewerk mit genialen Rezepten zu jedem Thema**
- **EXTRA: 700 Seiten Bonuskapitel zur GUI-Entwicklung mit WPF und Windows Forms**
- **Bonuskapitel sowie alle Beispiele aus dem Buch unter: www.doko-buch.de**

Dieser Klassiker der Visual Basic-/.NET-Programmierung bietet Ihnen Informationen und Rezepte, nach denen Sie anderswo vergeblich suchen. Einsteiger erhalten ein umfangreiches Praktikum zu den Grundlagen der .NET-Programmierung mit Visual Basic 2015, dem Profi liefert es fortgeschrittene Programmiertechniken zu allen wesentlichen Einsatzgebieten der Windowsprogrammierung. Zum sofortigen Ausprobieren finden Sie am Ende eines jeden Kapitels hochwertige Lösungen für nahezu jedes Problem.

Mit diesem Buch haben Sie den idealen Begleiter für Ihre tägliche Arbeit und zugleich – dank der erfrischenden und unterhaltsamen Sprache – eine durchaus spannende Lektüre auch für die Zeit nach der Arbeit.

Die Themenkomplexe im Buch:
Teil I: Grundlagen der Programmierung mit Visual Basic 2015
Teil II: Technologien
Teil III: Windows Apps

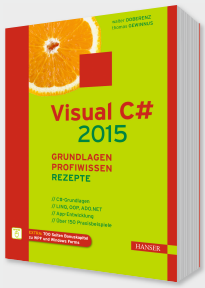
Die Themenkomplexe im Internet:
Teil IV: WPF-Anwendungen
Teil V: Windows Forms

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 //

- Grundlagen der Sprache
- OOP-Konzepte
- Einführung in LINQ
- Dateien & Dateisystem
- XML-Dateien
- Asynchrone Programmierung & Task Parallel Library
- Fehlersuche und Behandlung
- Datenbankzugriff mit ADO.NET
- Verteilen von Anwendungen
- App-Oberflächen entwerfen
- App-Entwicklung im Detail
- GUI-Entwicklung mit WPF und Windows Forms

UNSER BUCHTIPP FÜR SIE //



Doberenz/Gewinnus, Visual C# 2015
2015. 1200 Seiten. FlexCover. € 49,99.
ISBN 978-3-446-44381-5

HANSER

€ 49,99 [D] | € 51,40 [A]
ISBN 978-3-446-44380-8

9 783446 443808

www.hanser-fachbuch.de/computer